Trace-based Just-in-Time Type Specialization for Dynamic Languages

동적 언어를 위한 트레이스 기반 적시 타입 특수화

Andreas Gal*+, Brendan Eich*, Mike Shaver*, David Anderson*, David Mandelin*,
Mohammad R. Haghighat$, Blake Kaplan*, Graydon Hoare*, Boris Zbarsky*, Jason Orendorff*,
Jesse Ruderman*, Edwin Smith#, Rick Reitmaier#, Michael Bebenita+, Mason Chang+#, Michael Franz+

Mozilla Corporation*
{gal,brendan,shaver,danderson,dmandelin,mrbkap,graydon,bz,jorendorff,jruderman}@mozilla.com

Mozilla Corporation* {gal,brendan,shaver,danderson,dmandelin,mrbkap,graydon,bz,jorendorff,jruderman}@mozilla.com

Adobe Corporation#
{edwsmith,rreitmai}@adobe.com

Adobe Corporation# {edwsmith,rreitmai}@adobe.com

Intel Corporation$
{mohammad.r.haghighat}@intel.com

Intel Corporation$ 소속 {mohammad.r.

University of California, Irvine+
{mbebenit,changm,franz}@uci.edu

University of California, Irvine+ 소속 {mbebenit,changm,franz}@uci.edu

Abstract

초록

Dynamic languages such as JavaScript are more difficult to compile than statically typed ones. Since no concrete type information is available, traditional compilers need to emit generic code that can handle all possible type combinations at runtime. We present an alternative compilation technique for dynamically-typed languages that identifies frequently executed loop traces at run-time and then generates machine code on the fly that is specialized for the actual dynamic types occurring on each path through the loop. Our method provides cheap inter-procedural type specialization, and an elegant and efficient way of incrementally compiling lazily discovered alternative paths through nested loops. We have implemented a dynamic compiler for JavaScript based on our technique and we have measured speedups of 10x and more for certain benchmark programs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors — *Incremental compilers, code generation*.

General Terms Design, Experimentation, Measurement, Performance.

Keywords JavaScript, just-in-time compilation, trace trees.

Google Mail, Google Docs, Zimbra Collaboration Suite와 같은 브라우저 기반 생산성 애플리케이션의 애플리케이션 로직에 사용된다. JavaScript와 같은 동적 언어는 컴파일하기가 더 이 영역에서 원활한 사용자 경험을 제공하고 새로운 세대의 애플리케이션을 구현하려면 정적 타입 언어보다 어렵다. 구체적인 타입 정보를 사용할 수 없으므로 기존 컴파일러는 범용 코드를 생성해야 한다. 또한 가상 머신은 원활한 사용자 경험과 새로운 세대의 애플리케이션을 지원하기 위해 짧은 시작 시간과 높은 성능을 제공해야 한다. 이 코드는 런타임에 가능한 모든 타입 조합을 처리할 수 있어야 한다. 우리는 동적 타입 언어를 위한 대안적 컴파일 기법을 제시한다. 정적 타입 언어용 컴파일러는 효율적인 머신 코드를 생성하기 위해 타입 정보에 의존한다. 이 기법은 런타임에 자주 실행되는 루프 트레이스를 식별한 다음 즉석에서 머신 코드를 생성한다. JavaScript와 같은 동적 타입 프로그래밍 언어에서는 표현식의 타입이 런타임에 달라질 수 있다. 이는 컴파일러가 더 이상 루프의 각 경로에서 나타나는 실제 동적 타입에 맞게 코드를 특수화할 수 없음을 의미한다. 우리의 방법은 저비용의 프로시저 간 타입 특수화를 제공하며, 연산을 하나의 특정 타입에 작용하는 머신 명령어로 쉽게 변환할 수 있는 우아하고 효율적인 방법을 제공한다. 정확한 타입 정보가 없으면 컴파일러는 모든 타입을 처리할 수 있는 더 느린 범용 머신 코드를 생성해야 하며, 중첩 루프에서 지연 발견된 대체 경로를 점진적으로 컴파일한다. 우리는 가능한 타입 조합을 구현했다. 컴파일 시점의 정적 타입 추론으로 최적화된 머신 코드를 생성하는 데 필요한 타입 정보를 수집할 수는 있지만, 기존의 정적 분석은 매우 비싸다. 우리는 이 기법을 기반으로 JavaScript용 동적 컴파일러를 구현했으며, 일부 벤치마크 프로그램에서 10배 이상의 성능 향상을 측정했다. 따라서 웹 브라우저의 고도로 상호작용적인 환경에는 적합하지 않다. 범주 및 주제 기술어 D.3.4 [프로그래밍 언어]: 프로세서 — 점진적 컴파일러, 코드 생성.

우리는 빠른 컴파일 속도와 생성된 머신 코드의 뛰어난 성능을 양립시키는 동적 언어용 트레이스 기반 컴파일 기법을 제시한다. 일반 용어: 설계, 실험, 측정, 성능. 우리 시스템은 혼합 모드 실행 방식을 사용한다. 시스템은 빠르게 시작되는 바이트코드 인터프리터에서 JavaScript 실행을 시작한다.

1. Introduction

Dynamic languages such as JavaScript, Python, and Ruby, are popular since they are expressive, accessible to non-experts, and make deployment as easy as distributing a source file. They are used for small scripts as well as for complex applications. JavaScript, for example, is the de facto standard for client-side web programming and is used for the application logic of browser-based productivity applications such as Google Mail, Google Docs and Zimbra Collaboration Suite. In this domain, in order to provide a fluid user experience and enable a new generation of applications, virtual machines must provide a low startup time and high performance.

Compilers for statically typed languages rely on type information to generate efficient machine code. In a dynamically typed programming language such as JavaScript, the types of expressions may vary at runtime. This means that the compiler can no longer easily transform operations into machine instructions that operate on one specific type. Without exact type information, the compiler must emit slower generalized machine code that can deal with all potential type combinations. While compile-time static type inference might be able to gather type information to generate optimized machine code, traditional static analysis is very expensive and hence not well suited for the highly interactive environment of a web browser.

We present a trace-based compilation technique for dynamic languages that reconciles speed of compilation with excellent performance of the generated machine code. Our system uses a mixed-mode execution approach: the system starts running JavaScript in a fast-starting bytecode interpreter. As the program runs, the system identifies *hot* (frequently executed) bytecode sequences, records them, and compiles them to fast native code. We call such a sequence of instructions a *trace*.

Unlike method-based dynamic compilers, our dynamic compiler operates at the granularity of individual loops. This design choice is based on the expectation that programs spend most of their time in hot loops. Even in dynamically typed languages, we expect hot loops to be mostly *type-stable*, meaning that the types of values are invariant. (12) For example, we would expect loop counters that start as integers to remain integers for all iterations. When both of these expectations hold, a trace-based compiler can cover the program execution with a small number of type-specialized, efficiently compiled traces.

Each compiled trace covers one path through the program with one mapping of values to types. When the VM executes a compiled trace, it cannot guarantee that the same path will be followed or that the same types will occur in subsequent loop iterations.

프로그램이 실행되는 동안 시스템은 다음을 수행한다. 키워드: JavaScript, 적시 컴파일, 트레이스 트리. 핫한(자주 실행되는) 바이트코드 시퀀스를 식별하고 기록한 후 빠른 네이티브 코드로 컴파일한다. 우리는 그러한 시퀀스를 1. 서론 명령어 시퀀스를 트레이스라고 부른다. JavaScript, Python, Ruby와 같은 동적 언어는 표현력이 풍부하고 비전문가도 쉽게 사용할 수 있어 인기가 높다. 메서드 기반 동적 컴파일러와 달리 우리의 동적 컴파일러는 개별 루프 단위로 작동한다. 이러한 설계 선택은 소스 파일을 배포하는 것만큼 배포를 쉽게 한다. 이들은 작은 스크립트뿐 아니라 복잡한 애플리케이션에도 사용된다. 이러한 선택은 프로그램이 실행 시간 대부분을 핫 루프에서 보낸다는 예상을 바탕으로 한다. 예를 들어 JavaScript는 핫 루프에서 실행 시간 대부분을 보낸다. 동적 타입 언어에서도 핫 루프는 대부분 타입 안정적일 것으로 예상한다. 즉, 값의 타입이 변하지 않는다. 예를 들어 JavaScript는 클라이언트 측 웹 프로그래밍의 사실상 표준이다. (12)

예를 들어 정수로 시작하는 루프 카운터는 모든 반복에서 계속 정수로 유지될 것으로 예상한다.

이 두 예상이 모두 성립하면 트레이스 기반 컴파일러는 프로그램 실행을 포괄할 수 있다.

PLDI'09, June 15–20, 2009, Dublin, Ireland.

트레이스를 실행할 때 동일한 경로를 따를 것이라고 보장할 수 없으며 PLDI'09, 2009년 6월 15-20일, Dublin, Ireland.

후속 루프 반복에서 동일한 타입이 나타날 것이라고도 보장할 수 없다.

Hence, recording and compiling a trace *speculates* that the path and typing will be exactly as they were during recording for subsequent iterations of the loop.

Every compiled trace contains all the *guards* (checks) required to validate the speculation. If one of the guards fails (if control flow is different, or a value of a different type is generated), the trace exits. If an exit becomes hot, the VM can record a *branch trace* starting at the exit to cover the new path. In this way, the VM records a *trace tree* covering all the hot paths through the loop.

Nested loops can be difficult to optimize for tracing VMs. In a naïve implementation, inner loops would become hot first, and the VM would start tracing there. When the inner loop exits, the VM would detect that a different branch was taken. The VM would try to record a branch trace, and find that the trace reaches not the inner loop header, but the outer loop header. At this point, the VM could continue tracing until it reaches the inner loop header again, thus tracing the outer loop inside a trace tree for the inner loop. But this requires tracing a copy of the outer loop for every side exit and type combination in the inner loop. In essence, this is a form of unintended tail duplication, which can easily overflow the code cache. Alternatively, the VM could simply stop tracing, and give up on ever tracing outer loops.

We solve the nested loop problem by recording *nested trace trees*. Our system traces the inner loop exactly as the naïve version. The system stops extending the inner tree when it reaches an outer loop, but then it starts a new trace at the outer loop header. When the outer loop reaches the inner loop header, the system tries to call the trace tree for the inner loop. If the call succeeds, the VM records the call to the inner tree as part of the outer trace and finishes the outer trace as normal. In this way, our system can trace any number of loops nested to any depth without causing excessive tail duplication.

These techniques allow a VM to dynamically translate a program to nested, type-specialized trace trees. Because traces can cross function call boundaries, our techniques also achieve the effects of inlining. Because traces have no internal control-flow joins, they can be optimized in linear time by a simple compiler (10). Thus, our tracing VM efficiently performs the same kind of optimizations that would require interprocedural analysis in a static optimization setting. This makes tracing an attractive and effective tool to type specialize even complex function call-rich code.

We implemented these techniques for an existing JavaScript interpreter, SpiderMonkey. We call the resulting tracing VM *TraceMonkey*. TraceMonkey supports all the JavaScript features of SpiderMonkey, with a 2x-20x speedup for traceable programs.

This paper makes the following contributions:

- We explain an algorithm for dynamically forming trace trees to cover a program, representing nested loops as nested trace trees.
- We explain how to speculatively generate efficient type-specialized code for traces from dynamic language programs.
- We validate our tracing techniques in an implementation based on the SpiderMonkey JavaScript interpreter, achieving 2x-20x speedups on many programs.

The remainder of this paper is organized as follows. Section 3 is a general overview of trace tree based compilation we use to capture and compile frequently executed code regions. In Section 4 we describe our approach of covering nested loops using a number of individual trace trees. In Section 5 we describe our trace-compilation based speculative type specialization approach we use to generate efficient machine code from recorded bytecode traces. Our implementation of a dynamic type-specializing compiler for JavaScript is described in Section 6. Related work is discussed in Section 8. In Section 7 we evaluate our dynamic compiler based on a set of industry benchmarks. The paper ends with conclusions in Section 9 and an outlook on future work is presented in Section 10.

```
1 for (var i = 2; i < 100; ++i) {
2   if (!primes[i])
3     continue;
4   for (var k = i + i; i < 100; k += i)
5     primes[k] = false;
6 }
```

Figure 1. Sample program: sieve of Eratosthenes. `primes` is initialized to an array of 100 `false` values on entry to this code snippet.

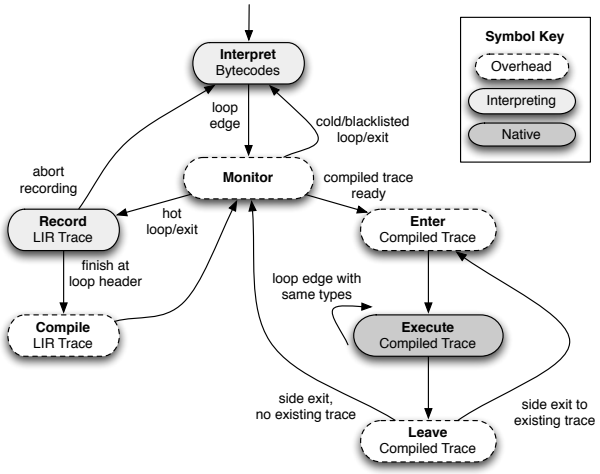


Figure 2. State machine describing the major activities of TraceMonkey and the conditions that cause transitions to a new activity. In the dark box, TM executes JS as compiled traces. In the light gray boxes, TM executes JS in the standard interpreter. White boxes are overhead. Thus, to maximize performance, we need to maximize time spent in the darkest box and minimize time spent in the white boxes. The best case is a loop where the types at the loop edge are the same as the types on entry–then TM can stay in native code until the loop is done.

2. Overview: Example Tracing Run

This section provides an overview of our system by describing how TraceMonkey executes an example program. The example program, shown in Figure 1, computes the first 100 prime numbers with nested loops. The narrative should be read along with Figure 2, which describes the activities TraceMonkey performs and when it transitions between the loops.

TraceMonkey always begins executing a program in the bytecode interpreter. Every loop back edge is a potential trace point. When the interpreter crosses a loop edge, TraceMonkey invokes the *trace monitor*, which may decide to record or execute a native trace. At the start of execution, there are no compiled traces yet, so the trace monitor counts the number of times each loop back edge is executed until a loop becomes *hot*, currently after 2 crossings. Note that the way our loops are compiled, the loop edge is crossed before entering the loop, so the second crossing occurs immediately after the first iteration.

Here is the sequence of events broken down by outer loop iteration:

```
v0 := ld state[748]     // load primes from the trace activation record
      st sp[0], v0      // store primes to interpreter stack
v1 := ld state[764]     // load k from the trace activation record
v2 := i2f(v1)           // convert k from int to double
      st sp[8], v1      // store k to interpreter stack
      st sp[16], 0      // store false to interpreter stack
v3 := ld v0[4]          // load class word for primes
v4 := and v3, -4        // mask out object class tag for primes
v5 := eq v4, Array      // test whether primes is an array
      xf v5             // side exit if v5 is false
v6 := js_Array_set(v0, v2, false)  // call function to set array element
v7 := eq v6, 0          // test return value from call
      xt v7             // side exit if js_Array_set returns false.
```

7절에서는 반복을 기준으로 활성화 레코드, 사이드 엑시트, 트레이스를 사용하는 동적 컴파일러를 평가한다: v0 := ld state[748] // load primes from the trace activation record st sp[0], v0 // store primes to interpreter stack v1 := ld state[764] // load k from the trace activation record v2 := i2f(v1) // convert k from int to double st sp[8], v1 // store k to interpreter stack st sp[16], 0 // store false to interpreter stack v3 := ld v0[4] // load class word for primes v4 := and v3, -4 // mask out object class tag for primes v5 := eq v4, Array // test whether primes is an array xf v5 // side exit if v5 is false v6 := js_Array_set(v0, v2, false) // call function to set array element v7 := eq v6, 0 // test return value from call xt v7 // side exit if js_Array_set returns false.

Figure 3. LIR snippet for sample program. This is the LIR recorded for line 5 of the sample program in Figure 1. The LIR encodes
그림 3. 예제 프로그램의 LIR 코드 조각. 이는 그림 1의 예제 프로그램에서 5번째 줄에 대해 기록된 LIR이다.
the semantics in SSA form using temporary variables. The LIR also encodes all the stores that the interpreter would do to its data stack.
LIR은 임시 변수를 사용하여 의미론을 SSA 형식으로 인코딩한다. LIR은 인터프리터가 데이터 스택에 수행할 모든 저장 연산도 인코딩한다.
Sometimes these stores can be optimized away as the stack locations are live only on exits to the interpreter. Finally, the LIR records guards
스택 위치는 인터프리터로 빠져나갈 때만 활성 상태이므로 이러한 저장 연산을 최적화로 제거할 수 있는 경우도 있다.
and side exits to verify the assumptions made in this recording: that `primes` is an array and that the call to set its element succeeds.

마지막으로 LIR은 이 기록에서 설정한 가정, 즉 primes이 배열이고 그 원소를 설정하는 호출이 성공한다는 가정을 검증하기 위한 가드와 사이드 엑시트를 기록한다.

```
mov edx, ebx(748)       // load primes from the trace activation record
mov edi(0), edx         // (*) store primes to interpreter stack
mov esi, ebx(764)       // load k from the trace activation record
mov edi(8), esi         // (*) store k to interpreter stack
mov edi(16), 0          // (*) store false to interpreter stack
mov eax, edx(4)         // (*) load object class word for primes
and eax, -4             // (*) mask out object class tag for primes
cmp eax, Array          // (*) test whether primes is an array
jne side_exit_1         // (*) side exit if primes is not an array
sub esp, 8              // bump stack for call alignment convention
push false              // push last argument for call
push esi                // push first argument for call
call js_Array_set       // call function to set array element
add esp, 8              // clean up extra stack space
mov ecx, ebx            // (*) created by register allocator
test eax, eax           // (*) test return value of js_Array_set
je side_exit_2          // (*) side exit if call failed
...
side_exit_1:
mov ecx, ebp(-4)        // restore ecx
mov esp, ebp            // restore esp
jmp epilog              // jump to ret statement
```

활성화 레코드, 사이드 엑시트, 트레이스는 다음과 같다: mov edx, ebx(748) // load primes from the trace activation record mov edi(0), edx // (*) store primes to interpreter stack mov esi, ebx(764) // load k from the trace activation record mov edi(8), esi // (*) store k to interpreter stack mov edi(16), 0 // (*) store false to interpreter stack mov eax, edx(4) // (*) load object class word for primes and eax, -4 // (*) mask out object class tag for primes cmp eax, Array // (*) test whether primes is an array jne side_exit_1 // (*) side exit if primes is not an array sub esp, 8 // bump stack for call alignment convention push false // push last argument for call push esi // push first argument for call call js_Array_set // call function to set array element add esp, 8 // clean up extra stack space mov ecx, ebx // (*) created by register allocator test eax, eax // (*) test return value of js_Array_set je side_exit_2 // (*) side exit if call failed ... side_exit_1: mov ecx, ebp(-4) // restore ecx mov esp, ebp // restore esp jmp epilog // jump to ret statement

---

Figure 4. x86 snippet for sample program. This is the x86 code compiled from the LIR snippet in Figure 3. Most LIR instructions compile
그림 4. 예제 프로그램의 x86 코드 조각. 이는 그림 3의 LIR 코드 조각에서 컴파일된 x86 코드이다.
to a single x86 instruction. Instructions marked with `(*)` would be omitted by an idealized compiler that knew that none of the side exits
대부분의 LIR 명령어는 단일 x86 명령어로 컴파일된다.
would ever be taken. The 17 instructions generated by the compiler compare favorably with the 100+ instructions that the interpreter would
어떤 사이드 엑시트도 발생하지 않을 것임을 아는 이상적인 컴파일러라면 (*)로 표시된 명령어를 생략할 것이다.
execute for the same code snippet, including 4 indirect jumps.

컴파일러가 생성한 17개 명령어는 인터프리터가 동일한 코드 조각에 대해 실행할 100개 이상의 명령어와 비교해 효율적이며, 후자에는 4개의 간접 점프가 포함된다.

i=2. This is the first iteration of the outer loop. The loop on
i=2. 이는 외부 루프의 첫 번째 반복이다.
lines 4-5 becomes hot on its second iteration, so TraceMonkey en-
인터프리터 PC의 루프와 값의 타입이 4-5번째 줄이 두 번째 반복에서 핫 상태가 되었을 때 관찰된 것과 일치하므로 TraceMonkey가 트레이스 기록을 시작했다.
ters recording mode on line 4. In recording mode, TraceMonkey
이 예제의 첫 번째 트레이스인 T45는 다음 줄에서 기록 모드로 진입한다.
records the code along the trace in a low-level compiler intermedi-
4. 기록 모드에서 TraceMonkey는 트레이스의 4번째 줄과 5. PC가 4번째 줄에 있고 k이 정수이며 primes이 객체이면 이 트레이스에 진입할 수 있으며, 트레이스를 따라 코드를 저수준 컴파일러 중간 표현으로 기록한다.
ate representation we call *LIR*. The LIR trace encodes all the oper-
T45를 컴파일한 후에는 이 표현을 LIR이라고 부른다.
ations performed and the types of all operands. The LIR trace also
LIR 트레이스는 수행된 모든 연산을 인코딩한다. TraceMonkey는 인터프리터로 돌아가 1번째 줄로 되돌아간다. 또한 모든 피연산자의 타입을 인코딩한다.
encodes *guards*, which are checks that verify that the control flow
LIR 트레이스도 이를 인코딩한다. i=3. 이제 1번째 줄의 루프 헤더가 핫 상태가 되었으므로 TraceMonkey가 트레이스 기록을 시작하며, 제어 흐름을 검증하는 검사인 가드를 인코딩한다.
and types are identical to those observed during trace recording.
기록이 4번째 줄에 도달하면, 트레이스 기록 중에 관찰된 것과 흐름 및 타입이 동일하다.
Thus, on later executions, if and only if all guards are passed, the
trace has the required program semantics.

TraceMonkey stops recording when execution returns to the
현재 트레이스 내부에 내부 루프를 중첩한다. 첫 번째 단계는 내부 트레이스를 서브루틴으로 호출하는 것이며, 실행이 트레이스로 돌아오면 TraceMonkey는 기록을 중지한다.
loop header or exits the loop. In this case, execution returns to the
이렇게 하면 4번째 줄의 루프가 완료될 때까지 실행한 뒤 루프 헤더로 돌아오거나 루프를 빠져나간다.
loop header on line 4.
이 경우 실행은 되돌아온 다음 기록기로 복귀한다. TraceMonkey는 트레이스 호출이 성공했는지 확인한 다음 4번째 줄의 루프 헤더에서 기록기로 돌아간다.

After recording is finished, TraceMonkey compiles the trace to
4. 호출이 성공했음을 확인한 다음 내부 트레이스 호출을 현재 트레이스의 일부로 기록한다. 기록이 끝나면 TraceMonkey는 트레이스를 현재 트레이스로 컴파일한다.
native code using the recorded type information for optimization.
실행이 해당 줄에 도달할 때까지 기록을 계속하며, 기록된 타입 정보를 최적화에 사용해 네이티브 코드로 컴파일한다.
The result is a native code fragment that can be entered if the
interpreter PC and the types of values match those observed when
trace recording was started. The first trace in our example, T_{45},
covers lines 4 and 5. This trace can be entered if the PC is at line 4,
`i` and `k` are integers, and `primes` is an object. After compiling T_{45},
TraceMonkey returns to the interpreter and loops back to line 1.

i=3. Now the loop header at line 1 has become hot, so Trace-
Monkey starts recording. When recording reaches line 4, Trace-
Monkey observes that it has reached an inner loop header that al-
ready has a compiled trace, so TraceMonkey attempts to nest the
inner loop inside the current trace. The first step is to call the inner
TraceMonkey는 이미 컴파일된 트레이스가 있는 내부 루프 헤더에 도달했음을 관찰한다. 따라서 이후 실행에서는 모든 가드를 통과하는 경우에만 트레이스가 필요한 프로그램 의미론을 가지므로, TraceMonkey는 현재 트레이스 안에 해당 트레이스를 중첩하려 한다.
trace as a subroutine. This executes the loop on line 4 to completion
and then returns to the recorder. TraceMonkey verifies that the call
was successful and then records the call to the inner trace as part of
the current trace. Recording continues until execution reaches line
1, and at which point TraceMonkey finishes and compiles a trace
for the outer loop, T_{16}.

i=4. On this iteration, TraceMonkey calls T_{16}. Because `i=4`, the `if` statement on line 2 is taken. This branch was not taken in the original trace, so this causes T_{16} to fail a guard and take a side exit. The exit is not yet hot, so TraceMonkey returns to the interpreter, which executes the continue statement.

i=5. TraceMonkey calls T_{16}, which in turn calls the nested trace T_{45}. T_{16} loops back to its own header, starting the next iteration without ever returning to the monitor.

i=6. On this iteration, the side exit on line 2 is taken again. This time, the side exit becomes hot, so a trace $T_{23,1}$ is recorded that covers line 3 and returns to the loop header. Thus, the end of $T_{23,1}$ jumps directly to the start of T_{16}. The side exit is patched so that on future iterations, it jumps directly to $T_{23,1}$.

At this point, TraceMonkey has compiled enough traces to cover the entire nested loop structure, so the rest of the program runs entirely as native code.

3. Trace Trees

In this section, we describe traces, trace trees, and how they are formed at run time. Although our techniques apply to any dynamic language interpreter, we will describe them assuming a bytecode interpreter to keep the exposition simple.

3.1 Traces

A *trace* is simply a program path, which may cross function call boundaries. TraceMonkey focuses on *loop traces*, that originate at a loop edge and represent a single iteration through the associated loop.

Similar to an extended basic block, a trace is only entered at the top, but may have many exits. In contrast to an extended basic block, a trace can contain join nodes. Since a trace always only follows one single path through the original program, however, join nodes are not recognizable as such in a trace and have a single predecessor node like regular nodes.

A *typed trace* is a trace annotated with a type for every variable (including temporaries) on the trace. A typed trace also has an entry *type map* giving the required types for variables used on the trace before they are defined. For example, a trace could have a type map `(x: int, b: boolean)`, meaning that the trace may be entered only if the value of the variable `x` is of type `int` and the value of `b` is of type `boolean`. The entry type map is much like the signature of a function.

In this paper, we only discuss typed loop traces, and we will refer to them simply as "traces". The key property of typed loop traces is that they can be compiled to efficient machine code using the same techniques used for typed languages.

In TraceMonkey, traces are recorded in trace-flavored SSA *LIR* (low-level intermediate representation). In trace-flavored SSA (or TSSA), phi nodes appear only at the entry point, which is reached both on entry and via loop edges. The important LIR primitives are constant values, memory loads and stores (by address and offset), integer operators, floating-point operators, function calls, and conditional exits. Type conversions, such as integer to double, are represented by function calls. This makes the LIR used by TraceMonkey independent of the concrete type system and type conversion rules of the source language. The LIR operations are generic enough that the backend compiler is language independent. Figure 3 shows an example LIR trace.

Bytecode interpreters typically represent values in a various complex data structures (e.g., hash tables) in a boxed format (i.e., with attached type tag bits). Since a trace is intended to represent efficient code that eliminates all that complexity, our traces operate on unboxed values in simple variables and arrays as much as possible.

A trace records all its intermediate values in a small activation record area. To make variable accesses fast on trace, the trace also imports local and global variables by unboxing them and copying them to its activation record. Thus, the trace can read and write these variables with simple loads and stores from a native activation recording, independently of the boxing mechanism used by the interpreter. When the trace exits, the VM boxes the values from this native storage location and copies them back to the interpreter structures.

For every control-flow branch in the source program, the recorder generates conditional exit LIR instructions. These instructions exit from the trace if required control flow is different from what it was at trace recording, ensuring that the trace instructions are run only if they are supposed to. We call these instructions *guard* instructions.

Most of our traces represent loops and end with the special `loop` LIR instruction. This is just an unconditional branch to the top of the trace. Such traces return only via guards.

Now, we describe the key optimizations that are performed as part of recording LIR. All of these optimizations reduce complex dynamic language constructs to simple typed constructs by specializing for the current trace. Each optimization requires guard instructions to verify their assumptions about the state and exit the trace if necessary.

Type specialization.

All LIR primitives apply to operands of specific types. Thus, LIR traces are necessarily type-specialized, and a compiler can easily produce a translation that requires no type dispatches. A typical bytecode interpreter carries tag bits along with each value, and to perform any operation, must check the tag bits, dynamically dispatch, mask out the tag bits to recover the untagged value, perform the operation, and then reapply tags. LIR omits everything except the operation itself.

A potential problem is that some operations can produce values of unpredictable types. For example, reading a property from an object could yield a value of any type, not necessarily the type observed during recording. The recorder emits guard instructions that conditionally exit if the operation yields a value of a different type from that seen during recording. These guard instructions guarantee that as long as execution is on trace, the types of values match those of the typed trace. When the VM observes a side exit along such a type guard, a new typed trace is recorded originating at the side exit location, capturing the new type of the operation in question.

Representation specialization: objects. In JavaScript, name lookup semantics are complex and potentially expensive because they include features like object inheritance and `eval`. To evaluate an object property read expression like `o.x`, the interpreter must search the property map of `o` and all of its prototypes and parents. Property maps can be implemented with different data structures (e.g., per-object hash tables or shared hash tables), so the search process also must dispatch on the representation of each object found during search. TraceMonkey can simply observe the result of the search process and record the simplest possible LIR to access the property value. For example, the search might finds the value of `o.x` in the prototype of `o`, which uses a shared hash-table representation that places `x` in slot 2 of a property vector. Then the recorded can generate LIR that reads `o.x` with just two or three loads: one to get the prototype, possibly one to get the property value vector, and one more to get slot 2 from the vector. This is a vast simplification and speedup compared to the original interpreter code. Inheritance relationships and object representations can change during execution, so the simplified code requires guard instructions that ensure the object representation is the same. In TraceMonkey, objects' rep-

1행에 도달하면 TraceMonkey는 외부 루프의 트레이스 T16을 완성하고 컴파일한다. 그 결과는 i=4일 때 진입할 수 있는 네이티브 코드 조각이다. 이 반복에서 TraceMonkey는 T16 트레이스를 호출한다.
i=4이므로 2행의 if 문을 실행한다. 트레이스는 모든 중간 값을 작은 활성화 레코드 영역에 기록한다.
이 분기는 기록 영역에서 실행되지 않았다. 트레이스에서 변수 접근을 빠르게 하기 위해, 트레이스는 지역 및 전역 변수를 언박싱하고 복사하여 가져온다. 이는 원래 트레이스와 다르므로 T16은 가드에서 실패하고 사이드 엑시트를 택한다.
이 트레이스의 엑시트는 아직 핫하지 않으므로 TraceMonkey는 인터프리터로 돌아가며, 그것들을 활성화 레코드로 복사한다.
따라서 트레이스는 이 변수들을 읽고 쓸 수 있으며, 인터프리터는 continue 문을 실행한다.
이 변수들은 네이티브 활성화 레코드에서 단순한 로드와 스토어로 접근할 수 있다. i=5이다.
TraceMonkey는 T16을 호출하고, T16은 다시 중첩 트레이스 T45를 호출한다. 이 기록은 T45가 사용하는 박싱 메커니즘과 독립적으로 이루어진다.
T16은 자체 헤더로 되돌아가 인터프리터로 복귀하지 않고 다음 반복을 시작한다. 트레이스가 종료되면 VM은 모니터로 돌아가지 않고 네이티브 저장 위치의 값들을 박싱한다.
이 값들을 네이티브 저장 위치에서 인터프리터로 다시 복사한다. i=6이다. 이 반복에서는 2행의 사이드 엑시트를 다시 택한다.
이는 이러한 구조를 형성한다. 이번에는 사이드 엑시트가 핫해지므로 3행을 포함하고 루프 헤더로 돌아가는 트레이스 T23,1을 기록한다. 소스 프로그램의 모든 제어 흐름 분기를 대상으로 한다.
따라서 T23,1의 끝에서 레코더는 조건부 엑시트 LIR 명령어를 생성한다. 이 명령어들은 T16의 시작점으로 직접 점프한다.
사이드 엑시트는 이후 반복에서 T23,1로 직접 점프하도록 패치된다. 필요한 제어 흐름이 트레이스 기록 당시와 다르면 이 명령어들이 트레이스에서 빠져나오게 한다.
이를 통해 트레이스 명령어는 실행되어야 할 때만 실행된다. 이 시점에 TraceMonkey는 전체 중첩 루프 구조를 포괄할 만큼 충분한 트레이스를 컴파일했다.
이러한 명령어를 가드 명령어라고 하며, 프로그램의 나머지 부분은 전체 중첩 루프 구조에서 실행된다.
전적으로 네이티브 코드로 실행된다.

대부분의 트레이스는 루프를 나타내며 특수한 loop LIR 명령어로 끝난다.

이는 단순히 트레이스의 시작점으로 향하는 무조건 분기이다. 이러한 트레이스는 가드를 통해서만 반환한다. 3. 트레이스 트리 이제 LIR 기록 과정에서 수행하는 핵심 최적화를 설명한다. 이 절에서는 트레이스와 트레이스 트리, 그리고 런타임에 이들이 형성되는 방식을 설명한다. 이러한 최적화는 모두 복잡한 동적 언어 구문을 단순한 타입 지정 구문으로 축소한다.
이 기법은 모든 동적 언어에 적용할 수 있지만, 여기서는 현재 트레이스에 맞게 특수화한다고 가정한다. 각 최적화에는 상태에 관한 가정을 검증하고 필요하면 종료하기 위한 가드 명령어가 필요하다. 설명을 단순하게 하기 위해 바이트코드 인터프리터를 가정한다.
필요하면 트레이스에서 빠져나온다.
타입 특수화. 3.1 트레이스 모든 LIR 프리미티브는 특정 타입의 피연산자에 적용된다. 따라서 트레이스는 함수 호출 경계를 넘을 수도 있는 하나의 프로그램 경로이며, LIR 트레이스는 필연적으로 타입 특수화된다. TraceMonkey는 루프 트레이스에 중점을 두며, 컴파일러는 타입 디스패치가 전혀 필요 없는 변환을 쉽게 생성할 수 있다. 루프 트레이스는 루프 에지에서 시작하여 해당 루프의 단일 반복을 나타낸다. 일반적인 바이트코드 인터프리터는 각 값에 태그 비트를 함께 저장한다. 연산을 수행하려면 태그 비트를 검사하고 동적으로 디스패치한 다음, 태그 비트를 마스킹하여 태그 없는 값을 복원해야 한다. 확장 기본 블록과 마찬가지로 트레이스에는 시작점으로만 진입하지만 여러 엑시트가 있을 수 있다. 확장 기본 블록과 달리, 연산을 수행한 다음 태그를 다시 적용한다. LIR은 연산 자체를 제외한 모든 것을 생략하며, 트레이스는 조인 노드를 포함할 수 있다. 하지만 트레이스는 항상 원래 프로그램의 단일 경로만 따른다. 잠재적인 문제는 일부 연산이 예측할 수 없는 타입의 값을 생성할 수 있다는 점이다. 따라서 조인 노드는 트레이스에서 별도로 식별되지 않고 일반 노드처럼 단일 선행 노드를 가지며, 일부 연산은 예측할 수 없는 타입의 값을 생성할 수 있다. 예를 들어 일반 노드처럼 선행 노드에서 프로퍼티를 읽는다. 객체는 어떤 타입의 값이든 산출할 수 있으며, 반드시 기록 중 관찰된 타입과 같지는 않다. 타입 지정 트레이스는 기록 중 관찰된 모든 변수에 타입이 주석으로 지정된 트레이스이다. 레코더는 트레이스에 가드 명령어(임시 변수 포함)를 방출한다. 타입 지정 트레이스에는 트레이스에서 사용되는 변수에 요구되는 타입을 제시하는 진입 타입 맵도 있으며, 연산이 기록 중 관찰된 것과 다른 타입의 값을 산출하면 조건부로 종료한다. 이러한 가드 명령어는 변수가 정의되기 전에 적용된다. 예를 들어 트레이스의 타입 맵은 (x: int, b: boolean)일 수 있으며, 이는 트레이스에 진입할 수 있음을 의미한다. 또한 실행이 트레이스에 머무는 동안 값의 타입이 타입 지정 트레이스의 타입과 일치함을 보장한다. VM이 변수 x의 값이 int 타입이고 b의 값이 boolean 타입인 경우에만 해당 타입 가드를 따라 사이드 엑시트를 관찰하면, 그 지점에서 시작하는 새 타입 지정 트레이스를 기록한다. 진입 타입 맵은 함수의 시그니처와 매우 유사하며, 사이드 엑시트 위치에서 해당 연산의 새 타입을 포착한다. 해당 연산이다. 이 논문에서는 타입 지정 루프 트레이스만 논의한다. 표현 특수화: 객체 JavaScript에서의 이름. 이하에서는 이를 간단히 '트레이스'라고 부른다. 타입 지정 루프 트레이스의 핵심 속성은 효율적인 기계어 코드로 컴파일할 수 있다는 점이다. 조회 의미론은 객체 상속과 eval 같은 기능을 포함하므로 복잡하고 잠재적으로 비용이 많이 든다. 타입 지정 언어에 사용되는 것과 동일한 기법을 평가하기 위해서이다. o.x 같은 객체 프로퍼티 읽기 표현식을 평가하려면 인터프리터는 o와 그 모든 프로토타입 및 부모의 프로퍼티 맵을 검색해야 한다. TraceMonkey에서 트레이스는 트레이스 지향 SSA LIR로 기록된다. (저수준 중간 표현). 트레이스 지향 SSA(또는 TSSA)에서 phi 노드는 진입점에만 나타나며, 이 지점에는 최초 진입과 루프 에지를 통해 모두 도달한다. 프로퍼티 맵은 서로 다른 자료구조(예: 객체별 해시 테이블 또는 공유 해시 테이블)로 구현할 수 있으므로 검색 방식도 달라질 수 있다. 트레이스에서 중요한 LIR 기본 연산에는 상숫값, 메모리 로드와 저장(주소 및 오프셋 기준), 정수 연산자, 부동소수점 연산자, 함수 호출, 조건부 엑시트가 있다. 검색 과정에서는 발견된 각 객체의 표현에 따라 디스패치해야 하지만, TraceMonkey는 검색 결과를 관찰하고 프로퍼티 값에 접근하는 가장 단순한 LIR를 기록할 수 있다. 프로퍼티 값에는 정수에서 double로의 변환 같은 타입 변환이 적용된다. 예를 들어 검색으로 찾은 값은 함수 호출로 표현될 수 있다. o의 프로토타입에 있는 o.x에는 x를 프로퍼티 벡터의 슬롯 2에 배치하는 공유 해시 테이블 표현이 사용된다. 이로써 TraceMonkey가 트레이스에 사용하는 LIR는 구체적인 타입 시스템과 타입 표현으로부터 독립적이 된다. 그런 다음 소스 언어의 변환 규칙이 기록된다. LIR 연산은 충분히 일반적이어서 백엔드 컴파일러가 언어에 독립적이며, o.x을 단 2번 또는 3번의 로드로 읽는 LIR를 생성할 수 있다. 프로토타입을 가져오는 데 1번, 필요하면 프로퍼티 값 벡터를 가져오는 데 1번이 사용된다. 그림 3은 LIR 트레이스의 예를 보여준다. 그리고 벡터의 슬롯 2에서 값을 가져오는 데 1번이 더 사용된다. 이는 기존 인터프리터 코드에 비해 대폭 단순화되고 빨라진 것이다. 바이트코드 인터프리터는 일반적으로 값을 다양한 형태로 표현한다. 상속 관계와 객체 표현은 실행 중 바뀔 수 있으며, 복잡한 자료구조(예: 해시 테이블)는 박싱 형식(즉, 타입 태그 비트가 부착된 형식)으로 표현된다. 트레이스는 이 모든 복잡성을 제거한 효율적인 코드를 나타내도록 의도되므로, 단순화된 코드는 객체 표현이 동일함을 보장하는 가드 명령어를 필요로 한다. TraceMonkey의 트레이스에서는 가능한 한 단순 변수와 배열의 언박싱 값을 대상으로 연산한다.

resentations are assigned an integer key called the *object shape*. Thus, the guard is a simple equality check on the object shape.

Representation specialization: numbers. JavaScript has no integer type, only a Number type that is the set of 64-bit IEEE-754 floating-pointer numbers ("doubles"). But many JavaScript operators, in particular array accesses and bitwise operators, really operate on integers, so they first convert the number to an integer, and then convert any integer result back to a double.[1] Clearly, a JavaScript VM that wants to be fast must find a way to operate on integers directly and avoid these conversions.

In TraceMonkey, we support two representations for numbers: integers and doubles. The interpreter uses integer representations as much as it can, switching for results that can only be represented as doubles. When a trace is started, some values may be imported and represented as integers. Some operations on integers require guards. For example, adding two integers can produce a value too large for the integer representation.

Function inlining. LIR traces can cross function boundaries in either direction, achieving function inlining. Move instructions need to be recorded for function entry and exit to copy arguments in and return values out. These move statements are then optimized away by the compiler using copy propagation. In order to be able to return to the interpreter, the trace must also generate LIR to record that a call frame has been entered and exited. The frame entry and exit LIR saves just enough information to allow the intepreter call stack to be restored later and is much simpler than the interpreter's standard call code. If the function being entered is not constant (which in JavaScript includes any call by function name), the recorder must also emit LIR to guard that the function is the same.

Guards and side exits. Each optimization described above requires one or more guards to verify the assumptions made in doing the optimization. A guard is just a group of LIR instructions that performs a test and conditional exit. The exit branches to a *side exit*, a small off-trace piece of LIR that returns a pointer to a structure that describes the reason for the exit along with the interpreter PC at the exit point and any other data needed to restore the interpreter's state structures.

Aborts. Some constructs are difficult to record in LIR traces. For example, `eval` or calls to external functions can change the program state in unpredictable ways, making it difficult for the tracer to know the current type map in order to continue tracing. A tracing implementation can also have any number of other limitations, e.g.,a small-memory device may limit the length of traces. When any situation occurs that prevents the implementation from continuing trace recording, the implementation *aborts* trace recording and returns to the trace monitor.

3.2 Trace Trees

Especially simple loops, namely those where control flow, value types, value representations, and inlined functions are all invariant, can be represented by a single trace. But most loops have at least some variation, and so the program will take side exits from the main trace. When a side exit becomes hot, TraceMonkey starts a new *branch trace* from that point and patches the side exit to jump directly to that trace. In this way, a single trace expands on demand to a single-entry, multiple-exit *trace tree*.

This section explains how trace trees are formed during execution. The goal is to form trace trees during execution that cover all the hot paths of the program.

Starting a tree. Tree trees always start at loop headers, because they are a natural place to look for hot paths. In TraceMonkey, loop headers are easy to detect–the bytecode compiler ensures that a bytecode is a loop header iff it is the target of a backward branch. TraceMonkey starts a tree when a given loop header has been executed a certain number of times (2 in the current implementation). Starting a tree just means starting recording a trace for the current point and type map and marking the trace as the root of a tree. Each tree is associated with a loop header and type map, so there may be several trees for a given loop header.

Closing the loop. Trace recording can end in several ways. Ideally, the trace reaches the loop header where it started with the same type map as on entry. This is called a *type-stable* loop iteration. In this case, the end of the trace can jump right to the beginning, as all the value representations are exactly as needed to enter the trace. The jump can even skip the usual code that would copy out the state at the end of the trace and copy it back in to the trace activation record to enter a trace.

In certain cases the trace might reach the loop header with a different type map. This scenario is sometime observed for the first iteration of a loop. Some variables inside the loop might initially be *undefined*, before they are set to a concrete type during the first loop iteration. When recording such an iteration, the recorder cannot link the trace back to its own loop header since it is *type-unstable*. Instead, the iteration is terminated with a side exit that will always fail and return to the interpreter. At the same time a new trace is recorded with the new type map. Every time an additional type-unstable trace is added to a region, its exit type map is compared to the entry map of all existing traces in case they complement each other. With this approach we are able to cover type-unstable loop iterations as long they eventually form a stable equilibrium.

Finally, the trace might exit the loop before reaching the loop header, for example because execution reaches a `break` or `return` statement. In this case, the VM simply ends the trace with an exit to the trace monitor.

As mentioned previously, we may speculatively chose to represent certain Number-typed values as integers on trace. We do so when we observe that Number-typed variables contain an integer value at trace entry. If during trace recording the variable is unexpectedly assigned a non-integer value, we have to widen the type of the variable to a double. As a result, the recorded trace becomes inherently type-unstable since it starts with an integer value but ends with a double value. This represents a mis-speculation, since at trace entry we specialized the Number-typed value to an integer, assuming that at the loop edge we would again find an integer value in the variable, allowing us to close the loop. To avoid future speculative failures involving this variable, and to obtain a type-stable trace we note the fact that the variable in question as been observed to sometimes hold non-integer values in an advisory data structure which we call the "oracle".

When compiling loops, we consult the oracle before specializing values to integers. Speculation towards integers is performed only if no adverse information is known to the oracle about that particular variable. Whenever we accidentally compile a loop that is type-unstable due to mis-speculation of a Number-typed variable, we immediately trigger the recording of a new trace, which based on the now updated oracle information will start with a double value and thus become type stable.

Extending a tree. Side exits lead to different paths through the loop, or paths with different types or representations. Thus, to completely cover the loop, the VM must record traces starting at all side exits. These traces are recorded much like root traces: there is a counter for each side exit, and when the counter reaches a hotness threshold, recording starts. Recording stops exactly as for the root trace, using the loop header of the root trace as the target to reach.

[1] Arrays are actually worse than this: if the index value is a number, it must be converted from a double to a string for the property access operator, and then to an integer internally to the array implementation.

표현에는 객체 형태라고 하는 정수 키가 할당된다.
Thus, the guard is a simple equality check on the object shape.
트리 시작. 트레이스 트리는 항상 루프 헤더에서 시작한다. 따라서 가드는 객체 형태에 대한 단순한 동등성 검사이다.
루프 헤더는 빈번하게 실행되는 경로를 찾기에 자연스러운 지점이다. TraceMonkey에서는 루프 트레이스를 사용한다. 표현 특수화: 숫자.
루프 헤더는 쉽게 탐지할 수 있다. 바이트코드 컴파일러는 바이트코드가 역방향 분기의 대상인 경우에만 루프 헤더가 되도록 보장한다. JavaScript에는 정수 타입이 없으며, 64-bit IEEE 숫자의 집합인 Number 타입만 있다.
754 부동소수점 수("double")이다.
그러나 많은 JavaScript 연산자, 특히 배열 접근과 비트 단위 연산자는 실제로 정수를 대상으로 동작한다. TraceMonkey는 특정 루프 헤더가 일정 횟수만큼 실행되면 트레이스 트리를 시작한다(현재 구현에서는 2회).
따라서 먼저 숫자를 정수로 변환한다. 트리를 시작한다는 것은 현재 지점과 타입 맵에 대한 트레이스 기록을 시작하고, 해당 트레이스를 트리의 루트로 표시한다는 의미일 뿐이다.
각 트리는 정수 결과를 다시 double로 변환한다.1 트리는 루프 헤더 및 타입 맵과 연관되므로 특정 루프 헤더에 여러 트리가 존재할 수 있다. 빠른 실행을 목표로 하는 JavaScript VM은 이를 처리할 방법을 찾아야 한다.
정수를 직접 연산하여 이러한 변환을 피한다. 루프 닫기. 트레이스 기록은 여러 방식으로 종료될 수 있다.
TraceMonkey는 숫자에 대해 정수와 double이라는 2가지 표현을 지원한다. 이상적으로는 트레이스가 시작할 때와 동일한 타입 맵을 가진 채 시작 지점의 루프 헤더에 도달한다.
인터프리터는 가능한 한 정수 표현을 사용하며, 진입 시와 동일한 타입 맵을 유지한다.
이를 타입 안정적 루프 반복이라고 하며, double로만 표현할 수 있는 결과에 대해서는 표현을 전환한다.
이 경우 트레이스의 끝은 double 표현으로 곧바로 점프할 수 있다. 트레이스가 시작되면 일부 값을 가져와 시작 지점에서 정수로 표현할 수 있는데, 모든 값 표현이 필요한 형태와 일치하기 때문이다.
일부 정수 연산에는 트레이스 진입을 위한 가드가 필요하다.
이 점프는 일반적인 상태 복사 코드를 건너뛸 수도 있다. 예를 들어 2개의 정수를 더하면 정수 표현으로 나타내기에는 너무 큰 값이 생성될 수 있다. 이때 트레이스 끝에서 상태를 내보낸 뒤 다시 복사해 넣어야 한다.
트레이스에 진입하기 위한 트레이스 활성화 레코드이다.
함수 인라이닝. LIR 트레이스는 양방향으로 함수 경계를 넘을 수 있어 함수 인라이닝을 달성한다. 어떤 경우에는 트레이스가 서로 다른 타입 맵을 가진 채 루프 헤더에 도달할 수 있다.
이동 명령에는 서로 다른 타입 맵이 사용된다.
이 시나리오는 때때로 루프의 첫 번째 반복에서 관찰된다. 함수 진입과 종료 시 인수를 복사하기 위한 명령을 기록해야 한다.
루프 내부의 일부 변수는 처음에는 값을 반환하기 위해 존재할 수 있다. 이러한 이동문은 변수가 첫 번째 루프에서 구체적인 타입으로 설정되기 전에는 undefined이지만, 이후 컴파일러가 복사 전파를 사용해 최적화하여 제거한다.
반복을 수행할 수 있도록 한다.
이러한 반복을 기록할 때 기록기는 트레이스가 타입 불안정 상태이므로 이를 자체 루프 헤더에 다시 연결할 수 없다. 인터프리터로 돌아가기 위해 트레이스는 LIR도 생성해야 한다.
호출 프레임에 진입하고 호출 프레임에서 빠져나왔음을 기록한다. 대신 반복은 항상 실패하여 인터프리터로 돌아가는 사이드 엑시트로 종료된다. 프레임 진입 및 종료 LIR은 나중에 복원할 수 있을 만큼의 정보만 저장한다.
동시에 새 타입 맵을 사용해 새 트레이스를 기록한다. 이 타입 맵은 인터프리터 호출 스택을 나중에 복원할 수 있게 하며 훨씬 단순하다.
타입 불안정 트레이스가 추가될 때마다 인터프리터의 표준 호출 코드가 사용된다. 타입 불안정 트레이스가 영역에 추가되면 서로 보완하는지 확인하기 위해 종료 타입 맵을 모든 기존 트레이스의 진입 맵과 비교한다. 진입하는 함수가 상수가 아닌 경우(JavaScript에서는 함수 이름을 통한 모든 호출이 이에 해당함), 기록기는 호출되는 함수가 동일한지 확인하는 가드용 LIR도 방출해야 한다.
이 접근법을 사용하면 동일한 타입 불안정 루프를 처리할 수 있다. 반복들이 궁극적으로 안정적인 평형을 형성하기만 하면 된다.
가드와 사이드 엑시트. 앞서 설명한 각 최적화에는 최적화 과정에서 세운 가정을 검증하기 위한 1개 이상의 가드가 필요하다. 마지막으로 실행이 break 또는 return에 도달하는 경우처럼 트레이스가 루프 헤더에 도달하기 전에 루프를 빠져나갈 수도 있다.
가드는 단지 LIR 명령문 그룹이다.
이 경우 VM은 테스트와 조건부 엑시트를 수행하는 엑시트로 트레이스를 끝내기만 한다. 이 엑시트는 트레이스 모니터로 분기한다.
사이드 엑시트는 트레이스 외부의 작은 LIR 조각으로, 엑시트 이유를 설명하는 구조체를 가리키는 포인터를 반환한다. 앞서 언급했듯이 트레이스에서는 특정 Number 타입 값을 정수로 표현하도록 추측에 기반해 선택할 수 있다.
이 구조체에는 엑시트 지점의 인터프리터 PC와 인터프리터의 상태 구조를 복원하는 데 필요한 기타 데이터가 포함된다. 트레이스 진입 시 Number 타입 변수에 정수 값이 들어 있는 것을 관찰하면 이를 수행한다.
트레이스 진입 시의 값이다.
이는 트레이스 진입 시 Number 타입 값을 정수로 특수화했으므로 잘못된 추측에 해당한다. 추적 구현에는 다른 제한이 얼마든지 있을 수 있으며, 예를 들어 메모리가 작은 장치에서는 트레이스 길이를 제한할 수 있다.
루프 가장자리에서 변수에 다시 정수 값이 들어 있어 루프를 닫을 수 있을 것이라고 가정한 것이다. 구현이 트레이스 기록을 계속하지 못하게 하는 상황이 발생할 수도 있다.
이 변수와 관련된 향후 추측 실패를 방지하고 타입이 안정적인 트레이스를 얻기 위해 관련 정보를 기록한다. 트레이스 기록을 계속할 수 없는 경우 구현은 기록을 중단하고 트레이스 모니터로 돌아간다.
해당 변수에서 정수가 아닌 값이 때때로 관찰되었다는 사실을 'oracle'이라 부르는 자문용 데이터 구조에 기록한다. 3.2 트레이스 트리
제어 흐름, 값 타입, 값 표현 및 인라인된 함수가 모두 불변인 특히 단순한 루프는 하나의 트레이스로 나타낼 수 있다. 루프를 컴파일할 때는 값을 정수로 특수화하기 전에 oracle을 참조한다.
정수에 대한 추측은 oracle에 해당 변수와 관련된 부정적인 정보가 없을 때만 수행하며, 이러한 루프는 하나의 트레이스로 나타낼 수 있다. 그러나 대부분의 루프에는 어느 정도 변형이 있으므로 프로그램은 메인 트레이스에서 사이드 엑시트로 빠져나간다. 정수에 대한 추측은 oracle에 해당 변수와 관련된 부정적인 정보가 없을 때만 수행한다.
Number 타입 변수에 대한 잘못된 추측으로 타입이 불안정한 루프를 실수로 컴파일하면 메인 트레이스에서 빠져나간다. 사이드 엑시트가 핫 상태가 되면 TraceMonkey는 그 지점에서 새로운 분기 트레이스를 시작하고, 해당 트레이스로 직접 점프하도록 사이드 엑시트를 패치한다. Number 타입 변수에 대한 잘못된 추측으로 타입이 불안정해지면 즉시 새 트레이스 기록을 시작한다. 이와 같이 하나의 트레이스는 필요에 따라 단일 진입점과 여러 엑시트를 갖는 트레이스 트리로 확장된다. 새 트레이스는 갱신된 oracle 정보에 따라 double 값으로 시작한다. 따라서 타입이 안정된다.
이 절에서는 실행 중 트레이스 트리가 형성되는 방식을 설명한다. 트리 확장.
사이드 엑시트는 루프를 통과하는 서로 다른 경로로 이어진다. 목표는 실행 중 프로그램의 모든 핫 경로를 포괄하는 트레이스 트리를 형성하는 것이며, 여기에는 루프를 통과하는 서로 다른 경로나 타입 또는 표현이 다른 경로가 포함된다.
따라서 프로그램의 핫 경로를 포괄해야 한다. 루프를 완전히 포괄하려면 VM이 모든 사이드 엑시트에서 시작하는 트레이스를 기록해야 한다.
이러한 트레이스는 루트 트레이스와 거의 같은 방식으로 기록된다. 각 사이드 엑시트에는 카운터가 있으며, 이 카운터가 핫니스 임계값에 도달하면 기록을 시작한다. 1 실제로 배열은 이보다 상황이 더 나쁘다. 인덱스 값이 숫자이면 프로퍼티 접근 연산자를 위해 변환해야 한다. 임계값에 도달하면 기록이 시작된다. 기록은 루트 트레이스와 정확히 같은 방식으로 중지된다. 프로퍼티 접근 연산자를 위해 double에서 문자열로 변환해야 하며, 도달할 대상으로 루트 트레이스의 루프 헤더를 사용한다. 그런 다음 배열 구현 내부에서 정수로 변환해야 한다.

Our implementation does not extend at all side exits. It extends only if the side exit is for a control-flow branch, and only if the side exit does not leave the loop. In particular we do not want to extend a trace tree along a path that leads to an outer loop, because we want to cover such paths in an outer tree through tree *nesting*.

3.3 Blacklisting

Sometimes, a program follows a path that cannot be compiled into a trace, usually because of limitations in the implementation. TraceMonkey does not currently support recording throwing and catching of arbitrary exceptions. This design trade off was chosen, because exceptions are usually rare in JavaScript. However, if a program opts to use exceptions intensively, we would suddenly incur a punishing runtime overhead if we repeatedly try to record a trace for this path and repeatedly fail to do so, since we abort tracing every time we observe an exception being thrown.

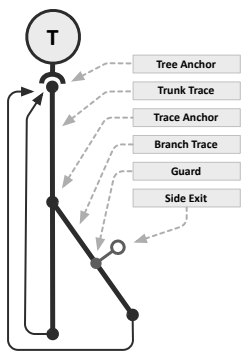


Figure 5. A tree with two traces, a trunk trace and one branch trace. The trunk trace contains a guard to which a branch trace was attached. The branch trace contain a guard that may fail and trigger a side exit. Both the trunk and the branch trace loop back to the tree anchor, which is the beginning of the trace tree.

우리 구현은 모든 사이드 엑시트에서 확장하지는 않는다. T는 사이드 엑시트가 제어 흐름 분기에 해당하고, 그 사이드 트리 앵커 엑시트가 루프를 벗어나지 않는 경우에만 확장한다. 특히 우리는 확장하려 하지 않는다. 주 트레이스 외부 루프로 이어지는 경로를 따라 트레이스 트리를 확장하지 않으려 한다. 그 이유는 트레이스 앵커 트리 중첩을 통해 외부 트리에서 그러한 경로를 처리하려 하기 때문이다. 분기 트레이스 3.3 블랙리스트 지정 가드 사이드 엑시트 때로는 대개 구현상의 한계 때문에 프로그램이 트레이스로 컴파일할 수 없는 경로를 따른다. 현재 TraceMonkey는 임의의 예외를 던지고 잡는 동작을 트레이스로 기록하는 기능을 지원하지 않는다. JavaScript에서는 대개 예외가 드물기 때문에 이러한 설계상의 절충안을 선택했다. 그러나 프로그램이 예외를 집중적으로 사용한다면, 이 경로의 트레이스를 반복해서 기록하려고 시도하다가 계속 실패할 경우 갑자기 막대한 런타임 오버헤드가 발생할 수 있다. 예외 발생을 관찰할 때마다 기록을 중단하기 때문이다. 그림 5. 주 트레이스 1개와 예외 발생이 관찰될 때마다 추적하는 분기 트레이스 1개, 총 2개의 트레이스로 이루어진 트리이다.

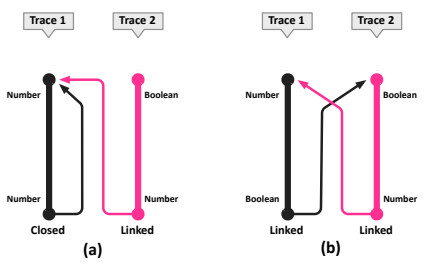
As a result, if a hot loop contains traces that always fail, the VM could potentially run much more slowly than the base interpreter: the VM repeatedly spends time trying to record traces, but is never able to run any. To avoid this problem, whenever the VM is about to start tracing, it must try to predict whether it will finish the trace.

트레이스이다. 주 트레이스에는 분기 트레이스가 연결된 가드가 포함되어 있다. 그 결과 핫 루프에 항상 실패하는 트레이스가 포함되어 있으면 VM은 분기 트레이스에는 실패하여 사이드 엑시트를 유발할 수 있는 가드가 포함되어 있다. 이 경우 기본 인터프리터보다 훨씬 느리게 실행될 가능성이 있다. 주 트레이스와 분기 트레이스는 모두 트레이스 트리의 시작점인 트리 앵커로 되돌아간다. VM은 트레이스 기록을 시도하는 데 반복해서 시간을 소비하지만 어떤 트레이스도 실행할 수 없다.

이 문제를 피하려면 VM은 트레이스 추적을 시작하려 할 때마다 트레이스를 완료할 수 있을지 예측해야 한다.

Our prediction algorithm is based on *blacklisting* traces that have been tried and failed. When the VM fails to finish a trace starting at a given point, the VM records that a failure has occurred. The VM also sets a counter so that it will not try to record a trace starting at that point until it is passed a few more times (32 in our implementation). This *backoff* counter gives temporary conditions that prevent tracing a chance to end. For example, a loop may behave differently during startup than during its steady-state execution. After a given number of failures (2 in our implementation), the VM marks the fragment as blacklisted, which means the VM will never again start recording at that point.

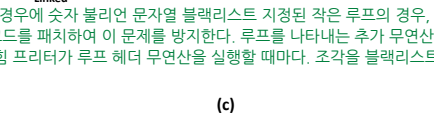
우리의 예측 알고리즘은 다음과 같은 트레이스를 블랙리스트 지정하는 데 기반한다



트레이스 1 트레이스 2 트레이스 1 트레이스 2 블랙리스트에 지정하는 방식에 기반한다.

VM이 특정 지점에서 시작한 트레이스를 완료하지 못하면 VM은 실패가 발생했음을 기록한다. 해당 숫자 불리언 숫자 불리언 VM은 또한 해당 지점을 몇 차례 더 통과하기 전까지는 그 지점에서 시작하는 트레이스를 기록하려 하지 않도록 카운터를 설정한다(우리 구현에서는 32회). 이 백오프 카운터는 트레이스 추적을 방해하는 일시적인 조건이 종료될 기회를 제공한다. 예를 들어 루프는 숫자 숫자 불리언 숫자 시작 단계에서는 정상 상태로 실행될 때와 다르게 동작할 수 있다. 일정- 닫힘 연결됨 연결됨 연결됨 횟수만큼 실패한 후(우리 구현에서는 2회), VM은 (b) (a) 해당 프래그먼트를 블랙리스트에 지정하며, 이는 VM이 절대로

After implementing this basic strategy, we observed that for small loops that get blacklisted, the system can spend a noticeable amount of time just finding the loop fragment and determining that it has been blacklisted. We now avoid that problem by patching the bytecode. We define an extra no-op bytecode that indicates a loop header. The VM calls into the trace monitor every time the interpreter executes a loop header no-op. To blacklist a fragment, we simply replace the loop header no-op with a regular no-op. Thus,



트레이스 1 트레이스 2 트레이스 3 그 지점에서 기록을 다시 시작하지 않는다. 이 기본 전략을 구현한 후, 우리는 다음의 경우에 숫자 불리언 문자열 블랙리스트 지정된 작은 루프의 경우, 시스템이 루프 조각을 찾아 그것이 블랙리스트 지정되었는지 확인하는 데만 상당한 시간을 소비할 수 있음을 관찰했다. 이제는 바이트코드를 패치하여 이 문제를 방지한다. 루프를 나타내는 추가 무연산 바이트코드를 정의한다. 문자열 숫자 문자열 헤더. VM은 인터프리터가 호출될 때마다 트레이스 모니터를 호출한다. 연결됨 연결됨 닫힘 프리터가 루프 헤더 무연산을 실행할 때마다. 조각을 블랙리스트 지정하기 위해, 우리는 연결됨

루프 헤더 무연산을 일반 무연산으로 바꾸기만 하면 된다. 따라서,

the interpreter will never again even call into the trace monitor.

인터프리터는 이후 트레이스 모니터를 호출조차 하지 않는다.

There is a related problem we have not yet solved, which occurs when a loop meets all of these conditions:

- The VM can form at least one root trace for the loop.
- There is at least one hot side exit for which the VM cannot complete a trace.
- The loop body is short.

In this case, the VM will repeatedly pass the loop header, search for a trace, find it, execute it, and fall back to the interpreter. With a short loop body, the overhead of finding and calling the trace is high, and causes performance to be even slower than the basic interpreter. So far, in this situation we have improved the implementation so that the VM can complete the branch trace. But it is hard to guarantee that this situation will never happen. As future work, this situation could be avoided by detecting and blacklisting loops for which the average trace call executes few bytecodes before returning to the interpreter.

(c)

Figure 6. We handle type-unstable loops by allowing traces to compile that cannot loop back to themselves due to a type mismatch. As such traces accumulate, we attempt to connect their loop edges to form groups of trace trees that can execute without having to side-exit to the interpreter to cover odd type cases. This is particularly important for nested trace trees where an outer tree tries to call an inner tree (or in this case a forest of inner trees), since inner loops frequently have initially undefined values which change type to a concrete value after the first iteration.

아직 해결하지 못한 관련 문제가 있으며, 이는 그림 6에서 발생한다. 우리는 타입 불안정 루프를 처리하기 위해 타입 불일치로 인해 자신으로 되돌아갈 수 없는 트레이스의 컴파일을 허용한다. 이는 루프가 다음 조건을 모두 충족할 때 발생한다. • VM은 해당 루프에 대해 적어도 1개의 루트 트레이스를 형성할 수 있다. 일치한다.

이러한 트레이스가 누적되면, 특이한 타입 사례를 처리하기 위해 인터프리터로 사이드 엑시트하지 않고도 실행할 수 있는 트레이스 트리 그룹을 형성하도록 루프 에지를 연결하려 한다. • VM이 트레이스를 완성할 수 없는 핫 사이드 엑시트가 적어도 1개 존재한다.

이는 트레이스를 부분적으로 완성한다. 이는 외부 트리가 내부 트리를 호출하려 하는 중첩 트레이스 트리에서 특히 중요하다. • 루프 본문은 짧다. 외부 트리가 내부 트리(또는 이 경우 내부 트리의 포리스트)를 호출하려 하기 때문이다. 내부 루프에는 처음에는 정의되지 않았다가 타입이 바뀌는 값이 흔히 존재한다. 이 경우 VM은 반복해서 루프 헤더를 지나 트레이스를 검색하고, 이를 찾아 실행한 뒤 인터프리터로 폴백한다.

4. Nested Trace Tree Formation

Figure 7 shows basic trace tree compilation (11) applied to a nested loop where the inner loop contains two paths. Usually, the inner loop (with header at i_2) becomes hot first, and a trace tree is rooted at that point. For example, the first recorded trace may be a cycle through the inner loop, $\{i_2, i_3, i_5, \alpha\}$. The α symbol is used to indicate that the trace loops back the tree anchor.

When execution leaves the inner loop, the basic design has two choices. First, the system can stop tracing and give up on compiling the outer loop, clearly an undesirable solution. The other choice is to continue tracing, compiling traces for the outer loop inside the inner loop's trace tree.

첫 번째 반복 후 구체적인 값으로 바뀐다. 루프 본문이 짧으면 트레이스를 찾아 호출하는 오버헤드가 크므로 기본 인터프리터보다도 성능이 느려진다. 내부 루프를 통과하는 경로는 {i2, i3, i5, α}이다. α 기호는 기본 인터프리터에 사용된다. 지금까지 이 상황에서는 트레이스가 트리 앵커로 되돌아간다는 것을 나타내도록 개선했다. VM이 분기 트레이스를 완성할 수 있도록 구현을 개선했다. 실행이 내부 루프를 벗어나면 기본 설계에는 2가지 선택지가 있다. 그러나 이러한 상황이 절대로 발생하지 않는다고 보장하기는 어렵다. 선택지가 있다. 첫째, 시스템은 추적을 중단하고 외부 루프의 컴파일을 포기할 수 있지만, 이는 분명 바람직하지 않은 해결책이다. 향후에는 이 상황을 감지하여 방지할 수 있다. 다른 선택지는 평균 트레이스 호출이 인터프리터로 돌아가기 전에 실행하는 바이트코드가 적은 루프를 블랙리스트 지정하는 한편, 추적을 계속하여 내부 루프 안에서 외부 루프용 트레이스를 컴파일하는 것이다. 내부 루프의 트레이스 트리.

For example, the program might exit at i_5 and record a branch trace that incorporates the outer loop: $\{i_5, i_7, i_1, i_6, i_7, i_1, \alpha\}$. Later, the program might take the other branch at i_2 and then exit, recording another branch trace incorporating the outer loop: $\{i_2, i_4, i_5, i_7, i_1, i_6, i_7, i_1, \alpha\}$. Thus, the outer loop is recorded and compiled twice, and both copies must be retained in the trace cache.

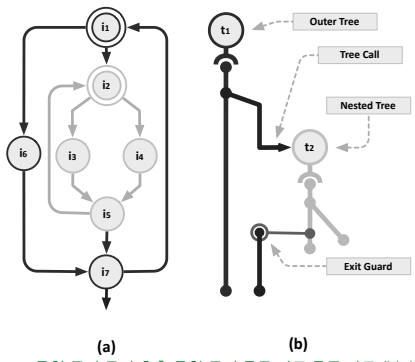
예를 들어, 프로그램은 i5에서 빠져나와 분기 트레이스를 기록할 수 있다. 4. 중첩 트레이스 트리 형성 외부 루프를 포함하는 트레이스는 {i5, i7, i1, i6, i7, i1, α}이다.

그림 7은 중첩 루프에 적용한 기본 트레이스 트리 컴파일 (11)을 보여준다. 이후 프로그램은 i2에서 다른 분기를 택한 다음, 내부 루프가 2개의 경로를 포함하는 루프를 실행할 수 있다.

보통 내부 루프(헤더는 i2)가 먼저 핫 상태가 되고 {i2, i4, i5, i7, i1, i6, i7, i1, α}에 루트를 둔 트레이스 트리가 생성되며, 내부 루프에서 이탈할 때 외부 루프를 포함하는 또 다른 분기 트레이스가 기록된다.

따라서 그 시점에 외부 루프가 기록된다.

예를 들어, 처음 기록된 트레이스는 두 번 컴파일되는 사이클일 수 있으며, 두 복사본 모두 트레이스 캐시에 유지해야 한다.



외부 트리 t1 i1 i1 t1 중첩 트리 트리 호출 중첩 트리 종료 가드 종료 가드 (b) (a)

그림 8.

Figure 7. Control flow graph of a nested loop with an if statement inside the inner most loop (a). An inner tree captures the inner loop, and is nested inside an outer tree which "calls" the inner tree. The inner tree returns to the outer tree once it exits along its loop condition guard (b).

2개의 중첩 루프가 있는 루프의 제어 흐름 그래프(왼쪽)와 그 중첩 트레이스 트리 구성(오른쪽).

외부 트리는 그림 7을 호출한다. if 문이 있는 중첩 루프의 제어 흐름 그래프는 2개의 내부 중첩 트레이스 트리를 호출하고, 가장 안쪽 루프 내부의 사이드 엑시트에 가드를 배치한다(a). 내부 트리는 내부 위치를 포착한다.

루프를 포착하며, 내부 트리를 '호출'하는 외부 트리 안에 중첩된다.

내부 트리는 루프 조건 가드를 따라 종료하면 외부 트리로 돌아간다(b).

In general, if loops are nested to depth k, and each loop has n paths (on geometric average), this naïve strategy yields $O(n^k)$ traces, which can easily fill the trace cache.

In order to execute programs with nested loops efficiently, a tracing system needs a technique for covering the nested loops with native code without exponential trace duplication.

4.1 Nesting Algorithm

The key insight is that if each loop is represented by its own trace tree, the code for each loop can be contained only in its own tree, and outer loop paths will not be duplicated. Another key fact is that we are not tracing arbitrary bytecodes that might have irreduceable control flow graphs, but rather bytecodes produced by a compiler for a language with structured control flow. Thus, given two loop edges, the system can easily determine whether they are nested and which is the inner loop. Using this knowledge, the system can compile inner and outer loops separately, and make the outer loop's traces *call* the inner loop's trace tree.

The algorithm for building nested trace trees is as follows. We start tracing at loop headers exactly as in the basic tracing system.

중첩 트레이스 트리를 구축하는 알고리즘은 다음과 같다. 4.2 중첩을 적용한 블랙리스트 지정. 기본 트레이싱 시스템과 정확히 동일하게 루프 헤더에서 트레이싱을 시작한다.

When we exit a loop (detected by comparing the interpreter PC with the range given by the loop edge), we stop the trace. The key step of the algorithm occurs when we are recording a trace for loop L_R (R for loop being recorded) and we reach the header of a different loop L_O (O for other loop). Note that L_O must be an inner loop of L_R because we stop the trace when we exit a loop.

블랙리스트 지정 알고리즘이 중첩과 원활하게 작동하도록 수정해야 한다. 루프를 벗어나면(인터프리터 PC를 루프 에지가 지정한 범위와 비교하여 감지한다) 트레이스를 중단한다. 문제는 외부 루프 트레이스가 시작 중 자주 중단된다는 점이다.

내부 트리를 사용할 수 없거나 내부 트리가 사이드 엑시트를 택하기 때문에 시작 중 중단되며, 이로 인해 기본 알고리즘에서는 외부 루프가 빠르게 블랙리스트 지정된다. 알고리즘의 핵심 단계는 LR(기록 중인 루프를 뜻하는 R)에 대한 트레이스를 기록하다가 다른 루프의 헤더에 도달할 때 발생한다.

- If L_O has a type-matching compiled trace tree, we call L_O as a nested trace tree. If the call succeeds, then we record the call in the trace for L_R. On future executions, the trace for L_R will call the inner trace directly.
- If L_O does not have a type-matching compiled trace tree yet, we have to obtain it before we are able to proceed. In order to do this, we simply abort recording the first trace. The trace monitor will see the inner loop header, and will immediately start recording the inner loop. [2]

그 다른 루프는 LO(다른 루프를 뜻하는 O)이다. 루프를 빠져나갈 때 트레이스를 중단하므로 LO는 LR의 내부 루프여야 한다는 점에 유의하라. 핵심 관찰은 내부 루프 때문에 외부 트레이스가 중단될 때

내부 트리가 준비되지 않은 상태라면 이는 일시적인 상황일 가능성이 높다는 것이다. • LO에 타입이 일치하는 컴파일된 트레이스 트리가 있으면 LO를 중첩 트레이스 트리로 호출한다. 따라서 이러한 중단은 다음 조건이 유지되는 한 블랙리스트 지정에 포함해서는 안 된다. 호출이 성공하면 내부 트리에 더 많은 트레이스를 구축할 수 있으므로 이 호출을 기록한다. LR의 트레이스에 기록한다. 이후 실행에서는 LR의 트레이스가 내부 트레이스를 직접 호출한다. 구현에서는 외부 트리가 내부 트리에서 중단되면 평소처럼 외부 트리의 블랙리스트 카운터를 증가시키고 컴파일 시도를 보류한다.

내부 트리가 트레이스를 완료하면 • LO에 타입이 일치하는 컴파일된 트레이스 트리가 아직 없는 경우 외부 루프의 블랙리스트 카운터를 감소시켜 이를 '용서'하며, 계속 진행하려면 먼저 해당 트레이스 트리를 확보해야 한다.

이는 외부 루프가 이전에 중단된 것을 보상하기 위한 것이다.

또한 이를 위해 백오프를 해제하며, 단순히 첫 번째 트레이스의 기록을 중단한다. 그러면 다음에 해당 지점에 도달할 때 외부 트리가 즉시 컴파일을 다시 시도할 수 있고, 트레이스 모니터는 내부 루프 헤더를 발견하자마자

If all the loops in a nest are type-stable, then loop nesting creates no duplication. Otherwise, if loops are nested to a depth k, and each loop is entered with m different type maps (on geometric average), then we compile $O(m^k)$ copies of the innermost loop. As long as m is close to 1, the resulting trace trees will be tractable.

An important detail is that the call to the inner trace tree must act like a function call site: it must return to the same point every time. The goal of nesting is to make inner and outer loops independent; thus when the inner tree is called, it must exit to the same point in the outer tree every time with the same type map. Because we cannot actually guarantee this property, we must guard on it after the call, and side exit if the property does not hold. A common reason for the inner tree not to return to the same point would be if the inner tree took a new side exit for which it had never compiled a trace. At this point, the interpreter PC is in the inner tree, so we cannot continue recording or executing the outer tree. If this happens during recording, we abort the outer trace, to give the inner tree a chance to finish growing. A future execution of the outer tree would then be able to properly finish and record a call to the inner tree. If an inner tree side exit happens during execution of a compiled trace for the outer tree, we simply exit the outer trace and start recording a new branch in the inner tree.

m이 1에 가까운 한, 결과로 생성되는 트레이스 트리는 다룰 수 있는 수준이다. 일반적으로 루프가 깊이 k까지 중첩되고 각 루프에 n개의 경로가 있다면, 내부 트레이스 트리 호출은 함수 호출 지점처럼 동작하여 매번 동일한 지점으로 반환되어야 한다는 점이 중요하다. 이 단순한 전략은 기하평균 기준으로 O(nk)개의 트레이스를 생성한다. 이는 트레이스 캐시를 쉽게 가득 채울 수 있다. 중첩의 목표는 내부 루프와 외부 루프를 독립적으로 만드는 것이다. 중첩 루프가 있는 프로그램을 효율적으로 실행하려면 트레이싱 시스템에 중첩 루프를 포괄하는 기법이 필요하다. 따라서 내부 트리가 호출될 때마다 동일한 타입 맵을 유지한 채 외부 트리의 동일한 지점으로 종료해야 한다. 이는 트레이스가 지수적으로 중복되지 않도록 네이티브 코드를 사용하기 때문이다. 실제로 이 속성을 보장할 수 없으므로 호출 후 이를 가드해야 하며, 속성이 성립하지 않으면 사이드 엑시트해야 한다. 4.1 중첩 알고리즘에서 내부 트리가 동일한 지점으로 반환되지 않는 일반적인 이유는, 아직 트레이스를 컴파일하지 않은 새로운 사이드 엑시트를 내부 트리가 택했기 때문이다. 핵심 통찰은 각 루프를 자체 트레이스 트리로 표현하면 각 루프의 코드를 해당 트리 안에만 둘 수 있다는 것이다. 이 시점에 인터프리터 PC가 내부 트리에 있으므로 내부 루프와 외부 루프의 경로는 중복되지 않는다. 또 다른 핵심 사실은 인터프리터 PC가 내부 트리에 있으므로 외부 트리의 기록이나 실행을 계속할 수 없다는 것이다. 우리는 축약 불가능한 제어 흐름 그래프를 가질 수 있는 임의의 바이트코드를 트레이싱하는 것이 아니라, 구조화된 제어 흐름을 갖는 언어용 컴파일러가 생성한 바이트코드를 트레이싱한다. 기록 중 이런 일이 발생하면 내부 트리가 성장을 마칠 기회를 얻도록 외부 트레이스를 중단한다. 이후 구조화된 제어 흐름을 갖는 언어의 외부 트리가 실행되면 정상적으로 완료되어 내부 트리 호출을 기록할 수 있다. 따라서 2개의 루프 에지가 주어지면 시스템은 두 루프가 중첩되어 있는지와 어느 것이 내부 트리인지를 쉽게 판별할 수 있으며, 이후 외부 트리는 정상적으로 완료되어 내부 트리 호출을 기록할 수 있다. 외부 트리를 위해 컴파일된 트레이스를 실행하는 중 내부 트리의 사이드 엑시트가 발생하면, 어느 것이 내부 루프인지 판별한 뒤 외부 트레이스를 종료한다. 이 정보를 이용해 시스템은 내부 루프와 외부 루프를 별도로 컴파일할 수 있으며, 외부 트레이스를 종료한 뒤 내부 트리에 새 분기를 기록하기 시작한다. 외부 루프의 트레이스가 내부 루프의 트레이스 트리를 호출한다.

4.2 Blacklisting with Nesting

The blacklisting algorithm needs modification to work well with nesting. The problem is that outer loop traces often abort during startup (because the inner tree is not available or takes a side exit), which would lead to their being quickly blacklisted by the basic algorithm.

The key observation is that when an outer trace aborts because the inner tree is not ready, this is probably a temporary condition. Thus, we should not count such aborts toward blacklisting as long as we are able to build up more traces for the inner tree.

In our implementation, when an outer tree aborts on the inner tree, we increment the outer tree's blacklist counter as usual and back off on compiling it. When the inner tree finishes a trace, we decrement the blacklist counter on the outer loop, "forgiving" the outer loop for aborting previously. We also undo the backoff so that the outer tree can start immediately trying to compile the next time we reach it.

5. Trace Tree Optimization

내부 루프 기록을 시작한다. 2 중첩 구조의 모든 루프가 타입 안정적이면 루프 중첩으로 인한 중복은 발생하지 않는다. 그렇지 않고 루프가 깊이 k까지 중첩되며 각 루프가 5. 트레이스 트리 최적화

This section explains how a recorded trace is translated to an optimized machine code trace. The trace compilation subsystem, NANOJIT, is separate from the VM and can be used for other applications.

이 절에서는 기록된 트레이스가 2 외부 기록을 중단하는 대신 원칙적으로 기록을 일시 중단할 수도 최적화된 기계어 코드 트레이스로 변환되는 방법을 설명한다. 트레이스 컴파일 하위 시스템인 있지만, 그러려면 구현에서 NANOJIT는 VM과 분리되어 있으며 다른

애플리케이션에도 사용할 수 있다.

[2] Instead of aborting the outer recording, we could principally merely suspend the recording, but that would require the implementation to be able to record several traces simultaneously, complicating the implementation, while saving only a few iterations in the interpreter.

여러 트레이스를 동시에 기록할 수 있어야 하므로 구현이 복잡해지는 반면, 인터프리터에서 절약되는 것은 몇 번의 반복뿐이다.

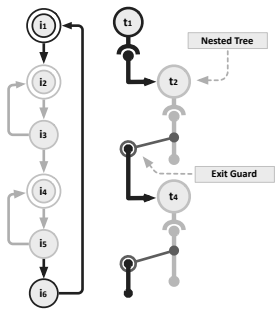


Figure 8. Control flow graph of a loop with two nested loops (left) and its nested trace tree configuration (right). The outer tree calls the two inner nested trace trees and places guards at their side exit locations.

5.1 Optimizations

Because traces are in SSA form and have no join points or ϕ-nodes, certain optimizations are easy to implement. In order to get good startup performance, the optimizations must run quickly, so we chose a small set of optimizations. We implemented the optimizations as pipelined filters so that they can be turned on and off independently, and yet all run in just two loop passes over the trace: one forward and one backward.

Tag	JS Type	Description
xx1	number	31-bit integer representation
000	object	pointer to JSObject handle
010	number	pointer to double handle
100	string	pointer to JSString handle
110	boolean	enumeration for null, undefined, true, false
	null, or undefined	

Figure 9. Tagged values in the SpiderMonkey JS interpreter. Testing tags, unboxing (extracting the untagged value) and boxing (creating tagged values) are significant costs. Avoiding these costs is a key benefit of tracing.

Every time the trace recorder emits a LIR instruction, the instruction is immediately passed to the first filter in the forward pipeline. Thus, forward filter optimizations are performed as the trace is recorded. Each filter may pass each instruction to the next filter unchanged, write a different instruction to the next filter, or write no instruction at all. For example, the constant folding filter can replace a multiply instruction like $v_{13} := mul3, 1000$ with a constant instruction $v_{13} = 3000$.

We currently apply four forward filters:

- On ISAs without floating-point instructions, a soft-float filter converts floating-point LIR instructions to sequences of integer instructions.
- CSE (constant subexpression elimination),
- expression simplification, including constant folding and a few algebraic identities (e.g., $a - a = 0$), and
- source language semantic-specific expression simplification, primarily algebraic identities that allow DOUBLE to be replaced with INT. For example, LIR that converts an INT to a DOUBLE and then back again would be removed by this filter.

When trace recording is completed, nanojit runs the backward optimization filters. These are used for optimizations that require backward program analysis. When running the backward filters, nanojit reads one LIR instruction at a time, and the reads are passed through the pipeline.

We currently apply three backward filters:

- Dead data-stack store elimination. The LIR trace encodes many stores to locations in the interpreter stack. But these values are never read back before exiting the trace (by the interpreter or another trace). Thus, stores to the stack that are overwritten before the next exit are dead. Stores to locations that are off the top of the interpreter stack at future exits are also dead.
- Dead call-stack store elimination. This is the same optimization as above, except applied to the interpreter's call stack used for function call inlining.
- Dead code elimination. This eliminates any operation that stores to a value that is never used.

After a LIR instruction is successfully read ("pulled") from the backward filter pipeline, nanojit's code generator emits native machine instruction(s) for it.

5.2 Register Allocation

We use a simple greedy register allocator that makes a single backward pass over the trace (it is integrated with the code generator). By the time the allocator has reached an instruction like $v_3 = add\ v_1, v_2$, it has already assigned a register to v_3. If v_1 and v_2 have not yet been assigned registers, the allocator assigns a free register to each. If there are no free registers, a value is selected for spilling. We use a class heuristic that selects the "oldest" register-carried value (6).

The heuristic considers the set R of values v in registers immediately after the current instruction for spilling. Let v_m be the last instruction before the current where each v is referred to. Then the heuristic selects v with minimum v_m. The motivation is that this frees up a register for as long as possible given a single spill.

If we need to spill a value v_s at this point, we generate the restore code just after the code for the current instruction. The corresponding spill code is generated just after the last point where v_s was used. The register that was assigned to v_s is marked free for the preceding code, because that register can now be used freely without affecting the following code

6. Implementation

To demonstrate the effectiveness of our approach, we have implemented a trace-based dynamic compiler for the SpiderMonkey JavaScript Virtual Machine (4). SpiderMonkey is the JavaScript VM embedded in Mozilla's Firefox open-source web browser (2), which is used by more than 200 million users world-wide. The core of SpiderMonkey is a bytecode interpreter implemented in C++.

In SpiderMonkey, all JavaScript values are represented by the type `jsval`. A `jsval` is machine word in which up to the 3 of the least significant bits are a type tag, and the remaining bits are data. See Figure 6 for details. All pointers contained in `jsvals` point to GC-controlled blocks aligned on 8-byte boundaries.

JavaScript *object* values are mappings of string-valued property names to arbitrary values. They are represented in one of two ways in SpiderMonkey. Most objects are represented by a shared structural description, called the *object shape*, that maps property names to array indexes using a hash table. The object stores a pointer to the shape and the array of its own property values. Objects with large, unique sets of property names store their properties directly in a hash table.

The garbage collector is an exact, non-generational, stop-the-world mark-and-sweep collector.

In the rest of this section we discuss key areas of the TraceMonkey implementation.

6.1 Calling Compiled Traces

Compiled traces are stored in a *trace cache*, indexed by intepreter PC and type map. Traces are compiled so that they may be called as functions using standard native calling conventions (e.g., `FASTCALL` on x86).

The interpreter must hit a loop edge and enter the monitor in order to call a native trace for the first time. The monitor computes the current type map, checks the trace cache for a trace for the current PC and type map, and if it finds one, executes the trace.

To execute a trace, the monitor must build a trace activation record containing imported local and global variables, temporary stack space, and space for arguments to native calls. The local and global values are then copied from the interpreter state to the trace activation record. Then, the trace is called like a normal C function pointer.

태그 JS 타입 설명: xx1 number 31비트 정수 표현, 000 object JSObject 핸들에 대한 포인터. 5.1 최적화. 트레이스는 SSA 형식이며 합류 지점이나 φ 노드가 없으므로 특정 최적화를 쉽게 구현할 수 있다. 010 number double 핸들에 대한 포인터, 100 string JSString 핸들에 대한 포인터. 우수한 시작 성능을 얻으려면 최적화가 빠르게 실행되어야 하므로 소규모 최적화 집합을 선택했다. 110 boolean은 null, undefined, true, false의 열거형이다. 최적화를 독립적으로 켜거나 끌 수 있으면서도 undefined 트레이스 전체를 순방향과 역방향으로 각각 한 번씩, 단 2번만 순회하도록 파이프라인 필터로 구현했다.

트레이스 기록기가 LIR 명령어를 내보낼 때마다 해당 명령어는 즉시 전달된다. 그림 9. SpiderMonkey JS 인터프리터의 태그된 값. 명령어는 즉시 순방향 파이프라인의 첫 번째 필터로 전달된다. 태그 검사, 언박싱(태그되지 않은 값 추출), 박싱 파이프라인이 이에 해당한다. 따라서 순방향 필터 최적화는 수행 과정에서 적용된다. (태그된 값 생성)은 상당한 비용을 유발한다. 이러한 비용을 피하면서 트레이스가 기록된다. 각 필터는 각 명령어를 다음 필터로 전달할 수 있으며, 이는 트레이싱의 핵심 이점이다. 필터는 명령어를 변경하지 않고 다음 필터에 전달하거나, 다른 명령어를 기록하거나, 아무 명령어도 기록하지 않을 수 있다.

예를 들어 상수 폴딩 필터는 v13 := mul3, 1000과 같은 곱셈 명령어를 상수 명령어 v13 = 3000으로 대체할 수 있다.

휴리스틱은 vm이 최소인 v을 선택한다. 그 동기는 단일 스필이 발생할 때 레지스터를 가능한 한 오래 비워 둘 수 있다는 점이다. 현재 4개의 순방향 필터를 적용한다.

이 지점에서 값 vs를 스필해야 한다면 현재 명령어의 코드 바로 뒤에 복원 코드를 생성한다. • 부동소수점 명령어가 없는 ISA에서는 소프트 플로트 필터를 사용한다. 이 필터는 부동소수점 LIR 명령어를 정수 명령어 시퀀스로 변환한다. 해당 스필 코드는 마지막 사용 지점 바로 뒤에 생성된다. vs가 사용된 지점이다. vs에 할당되었던 레지스터는 앞선 코드에서 사용 가능한 것으로 표시된다. 이제 그 레지스터를 자유롭게 사용해도 이후 코드에 영향을 주지 않기 때문이다. • CSE(상수 부분식 제거), • 상수 폴딩과 a 몇 가지 대수적 항등식(예: a –a = 0)을 포함하는 식 단순화, 그리고 6. 구현 • 소스 언어의 의미론에 특화된 식 단순화로, 주로 DOUBLE을 INT로 대체할 수 있게 하는 대수적 항등식을 적용한다. 접근 방식의 효과를 입증하기 위해 SpiderMonkey용 트레이스 기반 동적 컴파일러를 구현했다.

예를 들어 INT를 DOUBLE로 변환하는 LIR이 이에 해당한다. JavaScript VM (4).

SpiderMonkey는 JavaScript용이며, 이후 다시 원래 타입으로 변환하는 LIR은 이 필터에 의해 제거된다. Mozilla의 Firefox 오픈 소스 웹 브라우저에 내장된 VM이다 (2). 트레이스 기록이 완료되면 nanojit은 역방향 처리를 실행한다. Firefox는 전 세계적으로 200 million명 이상의 사용자가 이용한다. 핵심 최적화 필터. 이 필터들은 역방향 프로그램 분석이 필요한 최적화에 사용된다. SpiderMonkey의 핵심은 C++로 구현된 바이트코드 인터프리터이다. 역방향 프로그램 분석. 역방향 필터를 실행할 때 nanojit은 한 번에 하나의 LIR 명령어를 읽고, 읽은 명령어를 전달한다. SpiderMonkey에서는 모든 JavaScript 값이 jsval 타입으로 표현된다. jsval은 하위 최대 3비트가 타입 태그인 기계어 워드이며, 읽은 명령어는 파이프라인을 통과한다. 최하위 비트는 타입 태그이고 나머지 비트는 데이터이다. 현재 3개의 역방향 필터를 적용한다. 자세한 내용은 그림 6을 참조한다.

jsvals에 포함된 모든 포인터는 8-byte 경계에 정렬된 GC 관리 블록을 가리킨다. • 죽은 데이터 스택 저장 제거. LIR 트레이스는 인터프리터 스택 위치에 대한 많은 저장을 인코딩한다. JavaScript 객체 값은 문자열 값 속성을 매핑한 것이다. 그러나 이러한 값은 임의의 값에 대응하는 이름이다. 이 값들은 트레이스를 벗어나기 전에 인터프리터나 다른 트레이스에서 다시 읽히지 않는다. SpiderMonkey에서는 2가지 방식 중 하나로 표현되며, 대부분의 객체는 공유 구조 설명으로 표현된다. 따라서 다음 탈출 전에 덮어쓰이는 스택 저장은 죽은 저장이다. 객체 형태라 불리는 구조적 설명은 속성 이름을 매핑한다. 저장 위치는 해시 테이블을 사용하여 배열 인덱스에 매핑된다. 객체는 향후 탈출 시 인터프리터 스택의 최상단을 가리키는 포인터를 저장하며, 이러한 저장 역시 죽은 저장이다. 객체 형태와 객체 자체의 속성값 배열.

속성 이름 집합을 가진 객체. • 죽은 호출 스택 저장 제거.

이는 위와 동일한 최적화이지만 함수 호출 인라이닝에 사용되는 인터프리터의 호출 스택에 적용한다. 크고 고유한 속성 이름 집합은 속성을 해시 테이블에 직접 저장한다.

함수 호출 인라이닝.

가비지 컬렉터는 정확한 비세대형 stop-the-world 마크-스윕 컬렉터이다. • 죽은 코드 제거. 이는 결코 사용되지 않는 값에 저장하는 모든 연산을 제거한다. 이 절의 나머지에서는 트레이스 엔진인 TraceMonkey 구현의 핵심 영역을 논의한다.

핵심 구현.

LIR 명령어를 역방향 필터 파이프라인에서 성공적으로 읽어 오면("가져오면"), nanojit의 코드 생성기가 해당 명령어에 대응하는 네이티브 머신 명령어를 방출한다. 6.1 컴파일된 트레이스 호출. 컴파일된 트레이스는 인터프리터 PC와 타입 맵을 기준으로 인덱싱되는 트레이스 캐시에 저장된다. 트레이스는 표준 네이티브 호출 규약(예: x86의 FASTCALL)을 사용해 함수로 호출할 수 있도록 컴파일된다. 5.2 레지스터 할당에서는 한 차례만 순회하는 단순한 탐욕적 레지스터 할당기를 사용한다.

이 할당기는 트레이스를 역방향으로 순회하며 코드 생성기에 통합되어 있다. 인터프리터는 루프 에지에 도달해 모니터로 진입해야 한다.

할당기가 다음과 같은 명령어에 도달할 때쯤에는 네이티브 트레이스를 처음 호출하기 위한 단계에 이른다. 모니터가 v3 = add v1, v2를 계산할 때는 이미 v3에 레지스터를 할당한 상태이다.

v1과 v2에 아직 레지스터가 할당되지 않았다면 할당기는 빈 레지스터를 할당한다. 모니터는 현재 타입 맵을 계산하고 현재 PC와 타입 맵에 해당하는 트레이스를 트레이스 캐시에서 찾은 뒤, 발견하면 그 트레이스를 실행한다.

각각에 레지스터를 할당한다.

빈 레지스터가 없으면 스필할 값을 선택한다. 트레이스를 실행하려면 모니터가 트레이스 활성화 레코드를 구성해야 한다.

가장 오래전부터 레지스터에 실려 있던 값을 선택하는 클래스 휴리스틱을 사용한다 (6). 이 레코드에는 가져온 로컬 변수와 전역 변수 및 임시 값이 포함된다.

스택 공간과 네이티브 호출의 인수를 위한 공간이다. 로컬 변수와 전역 변수에 대해 휴리스틱은 현재 명령어 직후의 스필 대상을 고려한다. 로컬 및 전역 값은 인터프리터 상태에서 트레이스로 복사된다. 휴리스틱은 현재 명령어 직후 레지스터에 있는 값 v의 집합 R을 스필 대상으로 고려한다.

vm을 마지막 활성화 레코드라고 하자. 그런 다음 트레이스는 일반 C 함수처럼 호출된다. 각 v이 참조되는 현재 명령어 이전의 명령어이다.

그런 다음 포인터이다.

When a trace call returns, the monitor restores the interpreter state. First, the monitor checks the reason for the trace exit and applies blacklisting if needed. Then, it pops or synthesizes interpreter JavaScript call stack frames as needed. Finally, it copies the imported variables back from the trace activation record to the interpreter state.

트레이스 호출이 반환되면 모니터는 인터프리터 상태를 복원한다. 기록은 인터프리터 상태를 설정하는 포인터 교환으로 활성화된다. 먼저 모니터는 트레이스 종료 이유를 확인하고 필요하면 블랙리스트 지정을 적용한다. 인터프리터의 디스패치 테이블은 모든 바이트코드에 대해 하나의 '인터럽트' 루틴을 호출하도록 설정된다. 그런 다음 필요에 따라 인터프리터의 JavaScript 호출 스택 프레임을 제거하거나 합성한다. 인터럽트 루틴은 먼저 바이트코드별 기록 루틴을 호출한다. 마지막으로 가져온 변수를 복사한다. 그런 다음 필요하면 기록을 중단한다(예: 트레이스가 종료된 경우). 트레이스 활성화 레코드에서 가져온 변수는 인터프리터 상태로 다시 복사된다.

At least in the current implementation, these steps have a non-negligible runtime cost, so minimizing the number of interpreter-to-trace and trace-to-interpreter transitions is essential for performance. (see also Section 3.3). Our experiments (see Figure 12) show that for programs we can trace well such transitions happen infrequently and hence do not contribute significantly to total runtime. In a few programs, where the system is prevented from recording branch traces for hot side exits by aborts, this cost can rise to up to 10% of total execution time.

코드 구현이다. 일부 바이트코드는 실행 전에 예측할 수 없는 방식으로 타입 맵에 영향을 미친다(예: String.charCodeAt을 호출하면 인덱스 인수가 범위를 벗어날 경우 정수 또는 NaN을 반환한다). 적어도 현재 구현에서는 이러한 단계에 무시할 수 없는 런타임 비용이 들기 때문에 인터프리터에서 트레이스로, 그리고 트레이스에서 인터프리터로 전환하는 횟수를 최소화하는 것이 성능에 필수적이다. 이러한 경우에는 인터프리터가 적절히 처리하도록 구성한다. (3.3절도 참조한다). 실험 결과(그림 12 참조), 인터프리터는 바이트코드를 실행한 후 기록기를 다시 호출한다. 트레이스를 원활하게 수행할 수 있는 프로그램에서는 이러한 전환이 드물게 발생하므로 전체 런타임에 크게 영향을 미치지 않는다. 이러한 훅은 비교적 드물기 때문에 기록기가 현재 활성 상태인지 확인하는 추가 런타임 검사와 함께 인터프리터에 직접 삽입한다. 일부 프로그램에서는 시스템이 기록하지 못하게 되는 경우가 있으며, 이때 기록기가 현재 활성 상태인지 확인한다. 중단으로 인해 빈번한 사이드 엑시트의 분기 트레이스를 기록하지 못하면 이 비용이 전체 실행 시간의 최대 10%까지 증가할 수 있다. 인터프리터와 기록기를 분리하면 개별 코드의 복잡성은 줄어든다.

6.2 Trace Stitching

Transitions from a trace to a branch trace at a side exit avoid the costs of calling traces from the monitor, in a feature called *trace stitching*. At a side exit, the exiting trace only needs to write live register-carried values back to its trace activation record. In our implementation, identical type maps yield identical activation record layouts, so the trace activation record can be reused immediately by the branch trace.

In programs with branchy trace trees with small traces, trace stitching has a noticeable cost. Although writing to memory and then soon reading back would be expected to have a high L1 cache hit rate, for small traces the increased instruction count has a noticeable cost. Also, if the writes and reads are very close in the dynamic instruction stream, we have found that current x86 processors often incur penalties of 6 cycles or more (e.g., if the instructions use different base registers with equal values, the processor may not be able to detect that the addresses are the same right away).

The alternate solution is to recompile an entire trace tree, thus achieving inter-trace register allocation (10). The disadvantage is that tree recompilation takes time quadratic in the number of traces. We believe that the cost of recompiling a trace tree every time a branch is added would be prohibitive. That problem might be mitigated by recompiling only at certain points, or only for very hot, stable trees.

In the future, multicore hardware is expected to be common, making background tree recompilation attractive. In a closely related project (13) background recompilation yielded speedups of up to 1.25x on benchmarks with many branch traces. We plan to apply this technique to TraceMonkey as future work.

그러나 의미론적 동등성을 달성하려면 신중한 구현과 광범위한 테스트도 필요하다.

6.2 트레이스 연결. SpiderMonkey는 순수 인터프리터 성능에 유리한 것으로 밝혀진 팻 바이트코드 설계를 따르므로 이러한 동등성을 달성하기 어려운 경우가 있다. 사이드 엑시트에서 트레이스가 분기 트레이스로 전환되면 모니터에서 트레이스를 호출하는 비용을 피할 수 있다. 이를 트레이스 연결이라고 한다. 팻 바이트코드 설계에서는 개별 바이트코드가 복잡한 처리를 구현할 수 있다. 사이드 엑시트에서 종료되는 트레이스는 레지스터에 저장된 유효 값만 해당 트레이스 활성화 레코드에 다시 기록하면 된다. 구현상 동일한 타입 맵은 동일한 활성화 레코드 레이아웃을 생성하므로 분기 트레이스가 이를 즉시 재사용할 수 있다. 복잡한 처리의 예로는 캐시 및 밀집 배열 접근의 특수 사례를 포함하여 JavaScript 속성 값 접근 전체를 구현하는 getprop 바이트코드가 있다. 레이아웃이 동일하므로 분기 트레이스가 트레이스 활성화 레코드를 즉시 재사용할 수 있다. 팻 바이트코드에는 2가지 장점이 있다. 바이트코드 수가 적다. 따라서 디스패치 비용이 낮고, 바이트코드 구현이 클수록 컴파일러가 인터프리터를 최적화할 기회가 많아진다. 작은 트레이스가 많은 분기형 트레이스 트리를 사용하는 프로그램에서는 트레이스 연결에 눈에 띄는 비용이 발생한다. 트레이스 연결에는 눈에 띄는 비용이 발생한다. 또한 팻 바이트코드는 기록기가 동일한 특수 사례 로직을 같은 방식으로 다시 구현하도록 요구하므로 TraceMonkey에 문제가 된다. 메모리에 기록한 뒤 곧바로 다시 읽으면 L1 캐시 적중률이 높을 것으로 예상되지만, 작은 트레이스에서는 명령어 수 증가로 상당한 비용이 발생한다. 또한 팻 바이트코드는 기록기가 동일한 특수 사례 로직을 같은 방식으로 다시 구현하도록 요구하므로 TraceMonkey에 문제가 된다. 또한 다음과 같은 이유로 그 장점이 줄어든다. (a) 눈에 띄는 비용이 발생한다. 또한 기록과 읽기가 매우 근접해 있더라도 컴파일된 트레이스에서는 디스패치 비용이 완전히 제거된다. (b) 동적 명령어 스트림의 현재 트레이스에는 인터프리터의 거대한 코드 덩어리가 아니라 하나의 특수 사례만 포함된다. x86 프로세서에서는 6사이클 이상의 페널티가 발생하는 경우가 많다(예: (c) 트레이스 기반 TraceMonkey가 기본 인터프리터를 실행하는 데 소비하는 시간이 줄어든다. 명령어가 값은 같지만 서로 다른 베이스 레지스터를 사용하면 프로세서가 주소가 동일하다는 사실을 즉시 감지하지 못할 수 있다. 프로세서가 주소가 동일하다는 사실을 즉시 감지하지 못할 수 있다). 이러한 문제를 완화하기 위해 기록기에서 일부 복잡한 바이트코드를 단순한 바이트코드 시퀀스로 구현했다. 일부 복잡한 바이트코드를 기록기에서 단순한 바이트코드 시퀀스로 구현한다. 다른 해결책은 트레이스 트리 전체를 다시 컴파일하는 것이다. 원래의 의미론을 이러한 방식으로 표현하는 것은 그리 어렵지 않으며, 트레이스 간 레지스터 할당을 달성할 수 있다(10). 단점은 있지만 단순한 바이트코드를 기록하는 편이 훨씬 쉽다. 이를 통해 트레이스 트리를 다시 컴파일할 수 있지만, 재컴파일에는 트레이스 수의 제곱에 비례하는 시간이 든다. 이를 통해 팻 바이트코드의 장점을 유지하면서 트레이스 기록과 관련된 문제 일부를 피할 수 있다. 분기가 추가될 때마다 트레이스 트리를 다시 컴파일하는 비용은 지나치게 클 것으로 본다. 이는 인터프리터를 재귀적으로 다시 호출하는 팻 바이트코드에 특히 효과적이다. 분기가 추가될 때마다 재컴파일하는 비용은 지나치게 클 수 있다. 이 문제는 특정 시점에만 재컴파일하거나 매우 빈번하고 안정적인 트리만 재컴파일함으로써 완화할 수 있다. 예를 들어 객체의 잘 알려진 메서드를 호출하여 객체를 원시 값으로 변환할 수 있다. 객체의 메서드를 호출하는 경우 이 함수 호출을 인라인화할 수 있다. 향후 멀티코어 하드웨어가 보편화될 것으로 예상되므로 백그라운드 트리 재컴파일이 매력적인 선택지가 된다. 팻 연산 코드는 기록 중에만 더 작은 연산 코드로 분할한다는 점이 중요하다. 밀접하게 관련된 프로젝트에서는 기록 중에만 연산 코드를 다시 구성했다. 순수하게 인터프리터 방식으로 실행할 때(즉, 코드가 블랙리스트 지정된 경우) 인터프리터는 팻 연산 코드를 직접 효율적으로 실행한다. 관련 프로젝트 (13)에서는 백그라운드 재컴파일을 통해 분기 트레이스가 많은 벤치마크에서 최대 1.25배의 성능 향상을 얻었다. 우리는 fat opcode를 효율적으로 실행할 계획이다. 향후 작업으로 이 트레이스 기법을 TraceMonkey에 적용한다.

6.3 Trace Recording

The job of the trace recorder is to emit LIR with identical semantics to the currently running interpreter bytecode trace. A good implementation should have low impact on non-tracing interpreter performance and a convenient way for implementers to maintain semantic equivalence.

In our implementation, the only direct modification to the interpreter is a call to the trace monitor at loop edges. In our benchmark results (see Figure 12) the total time spent in the monitor (for all activities) is usually less than 5%, so we consider the interpreter impact requirement met. Incrementing the loop hit counter is expensive because it requires us to look up the loop in the trace cache, but we have tuned our loops to become hot and trace very quickly (on the second iteration). The hit counter implementation could be improved, which might give us a small increase in overall performance, as well as more flexibility with tuning hotness thresholds.

Once a loop is blacklisted we never call into the trace monitor for that loop (see Section 3.3).

6.3 트레이스 기록 6.4 선점 트레이스 기록기의 역할은 현재 실행 중인 인터프리터 바이트코드 트레이스와 의미가 동일한 LIR을 생성하는 것이다. 좋은 구현은 비트레이싱 인터프리터의 성능에 미치는 영향이 작아야 한다. 여러 VM과 마찬가지로 SpiderMonkey는 사용자 프로그램을 주기적으로 선점해야 한다. 주된 이유는 무한 루프에 빠진 스크립트가 호스트 시스템을 잠그는 것을 방지하고 GC를 예약하기 위해서이다. 또한 구현자가 의미론적 동등성을 편리하게 유지할 수 있어야 한다. 의미론적 동등성이다. 인터프리터에서는 모든 역방향 점프에서 확인하는 "지금 선점" 플래그를 설정하는 방식으로 이를 구현했다. 우리 구현에서 인터프리터에 직접 가한 유일한 수정 사항은 다음과 같다. 이는 루프 경계에서 트레이스 모니터를 호출하는 것이다. 이 전략은 TraceMonkey에도 이어졌다. VM은 모든 루프 경계에서 선점 플래그에 가드를 삽입한다. 벤치마크 결과(그림 12 참조), 모든 활동을 포함해 트레이스 모니터에서 소요된 총시간은 대체로 5% 미만이었다. 또한 이 추가 가드로 인한 실행 시간 증가는 대부분의 벤치마크에서 1% 미만이었다. 따라서 인터프리터에 미치는 영향에 관한 요건을 충족했다고 본다. 루프 적중 횟수 카운터를 증가시키려면 트레이스 캐시에서 루프를 조회해야 하므로 비용이 많이 든다. 실제로 이 비용을 감지할 수 있는 경우는 루프가 매우 짧은 프로그램뿐이다. 그러나 우리는 루프가 매우 빠르게 핫 상태가 되어 트레이스되도록 조정했다(두 번째 반복에서). 가드를 피하는 해결책도 시험했으나 채택하지 않았다. 적중 횟수 카운터의 구현을 개선하면 전반적인 성능이 조금 향상될 수 있다. 시험한 해결책은 루프 경계를 무조건 점프로 컴파일하고, 선점이 필요할 때 점프 대상을 종료 루틴으로 패치하는 방식이었다. 이렇게 하면 전반적인 성능이 향상될 뿐 아니라 핫니스 임계값을 조정할 때도 유연성이 커질 수 있다. 이 해결책은 일반적인 경우를 약간 더 빠르게 만들 수 있지만, 선점은 매우 느려진다. 이 해결책은 일반적인 경우를 약간 더 빠르게 만들 수 있지만, 선점은 매우 느려진다. 루프가 블랙리스트에 오르면 해당 루프에 대해서는 더 이상 트레이스 모니터를 호출하지 않는다. 해당 루프에 대해서는 그러하다(3.3절 참조). 구현도 매우 복잡했다.

6.4 Preemption

SpiderMonkey, like many VMs, needs to preempt the user program periodically. The main reasons are to prevent infinitely looping scripts from locking up the host system and to schedule GC.

In the interpreter, this had been implemented by setting a "preempt now" flag that was checked on every backward jump. This strategy carried over into TraceMonkey: the VM inserts a guard on the preemption flag at every loop edge. We measured less than a 1% increase in runtime on most benchmarks for this extra guard. In practice, the cost is detectable only for programs with very short loops.

We tested and rejected a solution that avoided the guards by compiling the loop edge as an unconditional jump, and patching the jump target to an exit routine when preemption is required. This solution can make the normal case slightly faster, but then preemption becomes very slow. The implementation was also very complex, especially trying to restart execution after the preemption.

특히 선점 후 실행을 다시 시작하는 과정이 복잡했다.

6.5 Calling External Functions

Like most interpreters, SpiderMonkey has a foreign function interface (FFI) that allows it to call C builtins and host system functions (e.g., web browser control and DOM access). The FFI has a standard signature for JS-callable functions, the key argument of which is an array of boxed values. External functions called through the FFI interact with the program state through an interpreter API (e.g., to read a property from an argument). There are also certain interpreter builtins that do not use the FFI, but interact with the program state in the same way, such as the `CallIteratorNext` function used with iterator objects. TraceMonkey must support this FFI in order to speed up code that interacts with the host system inside hot loops.

Calling external functions from TraceMonkey is potentially difficult because traces do not update the interpreter state until exiting. In particular, external functions may need the call stack or the global variables, but they may be out of date.

For the out-of-date call stack problem, we refactored some of the interpreter API implementation functions to re-materialize the interpreter call stack on demand.

We developed a C++ static analysis and annotated some interpreter functions in order to verify that the call stack is refreshed at any point it needs to be used. In order to access the call stack, a function must be annotated as either FORCESSTACK or REQUIRESSTACK. These annotations are also required in order to call REQUIRESSTACK functions, which are presumed to access the call stack transitively. FORCESSTACK is a trusted annotation, applied to only 5 functions, that means the function refreshes the call stack. REQUIRESSTACK is an untrusted annotation that means the function may only be called if the call stack has already been refreshed.

Similarly, we detect when host functions attempt to directly read or write global variables, and force the currently running trace to side exit. This is necessary since we cache and unbox global variables into the activation record during trace execution.

Since both call-stack access and global variable access are rarely performed by host functions, performance is not significantly affected by these safety mechanisms.

Another problem is that external functions can reenter the interpreter by calling scripts, which in turn again might want to access the call stack or global variables. To address this problem, we made the VM set a flag whenever the interpreter is reentered while a compiled trace is running.

Every call to an external function then checks this flag and exits the trace immediately after returning from the external function call if it is set. There are many external functions that seldom or never reenter, and they can be called without problem, and will cause trace exit only if necessary.

The FFI's boxed value array requirement has a performance cost, so we defined a new FFI that allows C functions to be annotated with their argument types so that the tracer can call them directly, without unnecessary argument conversions.

Currently, we do not support calling native property get and set override functions or DOM functions directly from trace. Support is planned future work.

Figure 11. Fraction of dynamic bytecodes executed by interpreter and on native traces. The speedup vs. interpreter is shown in parentheses next to each test. The fraction of bytecodes executed while recording is too small to see in this figure, except for `crypto-md5`, where fully 3% of bytecodes are executed while recording. In most of the tests, almost all the bytecodes are executed by compiled traces. Three of the benchmarks are not traced at all and run in the interpreter.

7. Evaluation

We evaluated our JavaScript tracing implementation using SunSpider, the industry standard JavaScript benchmark suite. SunSpider consists of 26 short-running (less than 250ms, average 26ms) JavaScript programs. This is in stark contrast to benchmark suites such as SpecJVM98 (3) used to evaluate desktop and server Java VMs. Many programs in those benchmarks use large data sets and execute for minutes. The SunSpider programs carry out a variety of tasks, primarily 3d rendering, bit-bashing, cryptographic encoding, math kernels, and string processing.

All experiments were performed on a MacBook Pro with 2.2 GHz Core 2 processor and 2 GB RAM running MacOS 10.5.

Benchmark results. The main question is whether programs run faster with tracing. For this, we ran the standard SunSpider test driver, which starts a JavaScript interpreter, loads and runs each program once for warmup, then loads and runs each program 10 times and reports the average time taken by each. We ran 4 different configurations for comparison: (a) SpiderMonkey, the baseline interpreter, (b) TraceMonkey, (d) SquirrelFish Extreme (SFX), the call-threaded JavaScript interpreter used in Apple's WebKit, and (e) V8, the method-compiling JavaScript VM from Google.

Figure 10 shows the relative speedups achieved by tracing, SFX, and V8 against the baseline (SpiderMonkey). Tracing achieves the best speedups in integer-heavy benchmarks, up to the 25x speedup on `bitops-bitwise-and`.

TraceMonkey is the fastest VM on 9 of the 26 benchmarks (`3d-morph`, `bitops-3bit-bits-in-byte`, `bitops-bitwise-and`, `crypto-sha1`, `math-cordic`, `math-partial-sums`, `math-spectral-norm`, `string-base64`, `string-validate-input`).

6.6 Correctness

During development, we had access to existing JavaScript test suites, but most of them were not designed with tracing VMs in mind and contained few loops.

One tool that helped us greatly was Mozilla's JavaScript fuzz tester, JSFUNFUZZ, which generates random JavaScript programs by nesting random language elements. We modified JSFUNFUZZ to generate loops, and also to test more heavily certain constructs we suspected would reveal flaws in our implementation. For example, we suspected bugs in TraceMonkey's handling of type-unstable loops and heavily branching code, and a specialized fuzz tester indeed revealed several regressions which we subsequently corrected.

6.5 외부 함수 호출 ?>9@AJ.D<F@-<>2.@A:0>#3$4,56# ? >9@AJ.0A:</C./8-2#3$4%56# 대부분의 인터프리터와 마찬가지로 SpiderMonkey에는 외부 함수 인터페이스가 >9@AJ.B<?><#3$4(56# 있으며(FFI), 이를 통해 C 내장 함수와 호스트 시스템 함수를 호출할 수 있다 ?>9@AJ.1<?2)'#3%4(56# (예: 웹 브라우저 제어 및 DOM 접근). FFI에는 표준 JS에서 호출 가능한 함수용 시그니처가 있으며, 핵심 인자는 박싱 값의 배열이다. 다음을 통해 호출되는 외부 함수는 FFI를 통해 인터프리터 API로 프로그램 상태와 상호작용한다(예: 인자에서 속성을 읽는 경우). 또한 특정 인터프리터 내장 함수는 FFI를 사용하지 않지만 프로그램 상태와 동일한 방식으로 상호작용하며, 그 예로 CallIteratorNext 함수가 있다 이 함수는 반복자 객체와 함께 사용된다. TraceMonkey는 트레이스에서 이 FFI를 지원해야 한다 핫 루프 내부에서 호스트 시스템과 상호작용하는 코드의 실행 속도를 높일 수 있다 1@>8:?.1@>?.@A.1=>2#3+4*56# 1@>8:?.&1@>.1@>?.@A.1=>2#3%(4(56# 루프. TraceMonkey의 트레이스에서 외부 함수를 호출하는 일은 잠재적으로 어렵다 트레이스가 종료되기 전까지 인터프리터 상태를 갱신하지 않기 때문이다 특히 외부 함수에는 호출 스택이나 전역 변수가 필요할 수 있지만, 이들은 최신 상태가 아닐 수 있다. 오래된 호출 스택 문제를 해결하기 위해, 우리는 일부를 리팩터링했다.

인터프리터 호출 스택을 필요할 때 다시 구체화하도록 인터프리터 API 구현 함수들을 수정했다. 호출 스택을 사용해야 하는 모든 지점에서 호출 스택이 갱신되는지 검증하기 위해 C++ 정적 분석을 개발하고 일부 인터프리터 함수에 주석을 지정했다. 호출 스택에 접근하기 위해, 그림 11. 동적 바이트코드 중 intera 함수가 실행한 비율에는 FORCESSTACK 또는 REpreter 주석을 지정해야 하며, 네이티브 트레이스에서도 마찬가지이다. 인터프리터 대비 성능 향상은 QUIRESSTACK으로 표시한다. 이러한 주석은 각 테스트 옆의 괄호 안에서 호출하기 위해서도 필요하다. 기록 중 실행되는 바이트코드의 비율은 이 그림에서 보이지 않을 정도로 작지만, 호출 스택에 전이적으로 접근한다고 간주되는 REQUIRESSTACK 함수는 예외이다. FORCESSTACK은 crypto-md5에 적용된 신뢰할 수 있는 주석으로, 바이트코드의 3%가 실행되는 동안 단 5개 함수에만 적용되며 해당 함수가 호출 스택을 갱신한다는 의미이다. 기록 중이다. 대부분의 테스트에서는 거의 모든 바이트코드가 컴파일된 트레이스에 의해 실행되며, REQUIRESSTACK은 함수에 관한 신뢰할 수 없는 주석이다. 벤치마크 중 3개는 트레이스되지 않으며, 해당 함수는 호출 스택이 이미 갱신된 경우에만 호출할 수 있다. 전혀 트레이스되지 않고 인터프리터에서 실행된다. 마찬가지로 호스트 함수가 전역 변수를 직접 읽거나 쓰려는 경우를 감지하여 현재 실행 중인 트레이스가 강제로 사이드 엑시트하도록 한다. 이는 트레이스 실행 중 전역 변수를 캐시하고 언박싱하여 활성화 레코드에 저장하기 때문에 필요하며, 루프와 분기가 많은 코드 및 특수화된 퍼즈 테스터도 관련된다. 실제로 퍼즈 테스터는 여러 회귀를 발견했으며, 이후 이를 수정했다. 호스트 함수가 호출 스택과 전역 변수에 접근하는 경우는 모두 드물기 때문에 성능에는 큰 영향을 미치지 않는다. 7. 평가 이러한 안전 메커니즘의 영향을 받는다.

또 다른 문제는 외부 함수가 스크립트를 호출하여 인터프리터에 재진입할 수 있고, 그 스크립트가 다시 접근을 시도할 수 있다는 점이다. 우리는 업계 표준 JavaScript 벤치마크 모음인 SunSpider를 사용하여 JavaScript 트레이싱 구현을 평가했다. SunSpi는 호출 스택 또는 전역 변수로 구성된다. 이 문제를 해결하기 위해 VM이 실행 시간이 짧은 26개(250ms 미만, 평균 26ms)의 JavaScript 프로그램을 실행하도록 했으며, 컴파일된 트레이스가 실행되는 동안 인터프리터에 재진입할 때마다 플래그를 설정하게 했다. 이는 컴파일된 트레이스가 실행 중인 벤치마크 모음과 현저한 대조를 이룬다. 예를 들어 데스크톱 및 서버 Java VM을 평가하는 데 사용되는 SpecJVM98 (3)이 있다. 외부 함수를 호출할 때마다 이 플래그를 확인하고 종료한다. 이러한 벤치마크의 많은 프로그램은 대규모 데이터 세트를 사용하고 수분 동안 실행되며, 외부 함수 호출에서 반환한 직후 트레이스를 종료한다. SunSpider 프로그램은 플래그가 설정된 경우 다양한 작업을 수행한다. 재진입하는 경우가 드물거나 전혀 없는 외부 함수가 많으며, 이러한 함수는 문제없이 호출할 수 있다. 주요 작업은 3d 렌더링, 비트 연산, 암호화 인코딩, 수학 커널, 문자열 처리이며 필요한 경우에만 종료를 유발한다. 필요한 경우에만 트레이스를 종료한다. 모든 실험은 2.2 GHz Core 2 프로세서와 2 GB RAM을 탑재하고 MacOS 10.5를 실행하는 MacBook Pro에서 수행했다. FFI의 박싱 값 배열 요구 사항에는 성능 비용이 따른다. 비용이 들기 때문에 C 함수에 인자 타입 주석을 지정할 수 있는 새로운 FFI를 정의했다. 벤치마크 결과. 주된 질문은 프로그램이 인자 타입으로 주석 처리되어 트레이서가 호출할 수 있을 때 트레이스를 사용하면 더 빠르게 실행되는지 여부이다. 이를 위해 불필요한 인자 변환 없이 표준 SunSpider 테스트를 직접 실행했다. 드라이버는 JavaScript 인터프리터를 시작하고 각 프로그램을 불러와 워밍업을 위해 1회 실행한 다음, 각 프로그램을 다시 불러와 10회 실행한다. 현재 네이티브 속성 get 및 set 재정의 함수나 DOM 함수를 트레이스에서 직접 호출하는 기능은 지원하지 않는다. 지원 횟수와 각 프로그램에 걸린 평균 시간을 보고한다. 향후 지원할 계획이며, 비교를 위해 서로 다른 4개 구성을 실행했다. 비교한 구성은 다음과 같다.

(a) 기준 인터프리터인 SpiderMonkey, (b) TraceMonkey, (d) Apple의 WebKit에서 사용되는 호출 스레드 방식의 JavaScript 인터프리터인 SquirrelFish Extreme(SFX). 6.6 정확성 개발 중에는 기존 JavaScript 테스트를 이용할 수 있었고,

(e) Google의 메서드 컴파일 방식 JavaScript VM인 V8. 테스트 스위트가 있었지만, 대부분은 트레이싱 VM을 염두에 두고 설계되지 않았고 루프도 거의 포함하지 않았다. 그림 10은 트레이싱과 SFX가 달성한 상대적 성능 향상을 보여준다.

그리고 V8을 기준 시스템(SpiderMonkey)과 비교한다. 트레이싱은 정수 연산 비중이 높은 벤치마크에서 가장 큰 성능 향상을 달성하며, bitops-bitwise-and에서는 최대 25배의 성능 향상을 보인다. 우리에게 큰 도움이 된 도구 중 하나는 무작위 JavaScript 프로그램을 생성하는 Mozilla의 JavaScript 퍼즈 테스터 JSFUNFUZZ였다.

무작위 언어 요소를 중첩하는 방식으로 생성한다.

우리는 JSFUNFUZZ가 루프를 생성하도록 수정하고, 트레이스 구현의 결함을 드러낼 것으로 예상한 특정 구문도 더 집중적으로 테스트하도록 했다. TraceMonkey는 26개 벤치마크 중 9개에서 가장 빠른 VM이다(3d-morph, bitops-3bit-bits-in-byte, bitops-bitwisewe, crypto-sha1, math-cordic, math-partial-sums, spectral-norm, string-base64, string-validate-input). and 예를 들어, 우리는 TraceMonkey가 타입이 불안정한 요소를 처리할 때 버그가 발생할 것으로 예상했다.

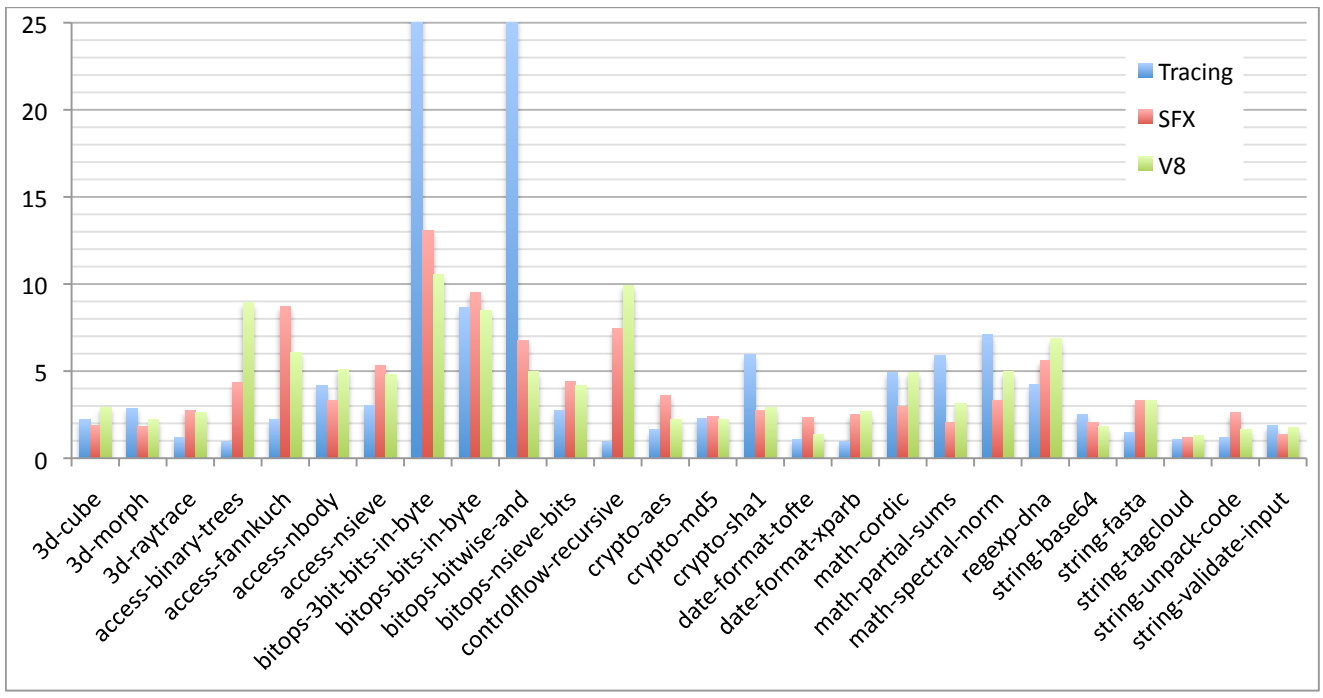


Figure 10. Speedup vs. a baseline JavaScript interpreter (SpiderMonkey) for our trace-based JIT compiler, Apple's SquirrelFish Extreme inline threading interpreter and Google's V8 JS compiler. Our system generates particularly efficient code for programs that benefit most from type specialization, which includes SunSpider Benchmark programs that perform bit manipulation. We type-specialize the code in question to use integer arithmetic, which substantially improves performance. For one of the benchmark programs we execute 25 times faster than the SpiderMonkey interpreter, and almost 5 times faster than V8 and SFX. For a large number of benchmarks all three VMs produce similar results. We perform worst on benchmark programs that we do not trace and instead fall back onto the interpreter. This includes the recursive benchmarks `access-binary-trees` and `control-flow-recursive`, for which we currently don't generate any native code.

그림 10.
기준 JavaScript 인터프리터(SpiderMonkey) 대비 본 논문의 트레이스 기반 JIT 컴파일러, Apple의 SquirrelFish Extreme 인라인 스레딩 인터프리터, Google의 V8 JS 컴파일러가 달성한 성능 향상.
본 시스템은 타입 특수화의 이점을 가장 크게 얻는 프로그램에 특히 효율적인 코드를 생성하며, 여기에는 비트 조작을 수행하는 SunSpider 벤치마크 프로그램이 포함된다.
해당 코드가 정수 산술을 사용하도록 타입 특수화를 적용하여 성능을 크게 향상시킨다.
벤치마크 프로그램 중 하나에서 본 시스템은 SpiderMonkey 인터프리터보다 25배 빠르고 V8 및 SFX보다 거의 5배 빠르게 실행된다.
많은 벤치마크에서 3개 VM은 모두 비슷한 결과를 보인다. 트레이스하지 않고 인터프리터로 대체 실행하는 벤치마크 프로그램에서 성능이 가장 낮다.
여기에는 재귀 벤치마크인 access-binary-trees와 control-flow-recursive가 포함되며, 현재 이들에 대해서는 네이티브 코드를 전혀 생성하지 않는다.

In particular, the `bitops` benchmarks are short programs that perform many bitwise operations, so TraceMonkey can cover the entire program with 1 or 2 traces that operate on integers. TraceMonkey runs all the other programs in this set almost entirely as native code.

특히 bitops 벤치마크는 많은 비트 단위 연산을 수행하는 짧은 프로그램이다.
많은 비트 단위 연산을 수행하므로 TraceMonkey는 전체를 포괄할 수 있으며, access-nbody는 많은 트레이스 (81)를 형성한다.
crypto-md5 전체 프로그램을 정수에 대해 연산하는 1개 또는 2개의 트레이스로 포괄한다.
TraceMonforms는 매우 긴 트레이스 1개를 형성한다. nanocode의 컴파일 속도를 개선하여 이 프로그램의 성능을 향상할 수 있을 것으로 예상하며, key는 이 집합의 다른 모든 프로그램을 거의 전적으로 네이티브 방식으로 실행한다.

`regexp-dna` is dominated by regular expression matching, which is implemented in all 3 VMs by a special regular expression compiler. Thus, performance on this benchmark has little relation to the trace compilation approach discussed in this paper.

TraceMonkey's smaller speedups on the other benchmarks can be attributed to a few specific causes:

- The implementation does not currently trace recursion, so TraceMonkey achieves a small speedup or no speedup on benchmarks that use recursion extensively: `3d-cube`, `3d-raytrace`, `access-binary-trees`, `string-tagcloud`, and `controlflow-recursive`.
- The implementation does not currently trace `eval` and some other functions implemented in C. Because `date-format-tofte` and `date-format-xparb` use such functions in their main loops, we do not trace them.
- The implementation does not currently trace through regular expression `replace` operations. The replace function can be passed a function object used to compute the replacement text.

Our implementation currently does not trace functions called as replace functions. The run time of `string-unpack-code` is dominated by such a `replace` call.

jit. regexp-dna는 정규식 일치 연산이 실행 시간의 대부분을 차지한다.
• 일부 프로그램은 트레이스가 매우 잘되고 인터프리터보다 빨라지지만, 3개 VM 모두에서 특수 정규식 컴파일러로 구현되므로 SFX 및 V8 중 하나 이상보다는 빠르지 않다. 따라서 이 벤치마크의 성능은 본 논문에서 설명하는 트레이스 컴파일 접근법 및 bitops-bits-in-byte, bitops-nsieve-bits, access와 거의 관련이 없다. fannkuch, access-nsieve, crypto-aes. 다른 벤치마크에서 TraceMonkey의 성능 향상이 더 작은 이유는 명확하지 않지만 몇 가지 구체적인 원인으로 설명할 수 있다. 이 프로그램들은 모두 본문이 작은 중첩 루프를 포함하므로, 구현에서 중첩 트레이스를 호출하는 비용이 비교적 높다고 추정한다.
string-fasta는 트레이스가 잘되지만 문자열 처리 내장 함수가 실행 시간의 대부분을 차지한다.
• 현재 구현은 재귀를 트레이스하지 않으며, 이러한 내장 함수는 트레이스의 영향을 받지 않고 SpiderMonkey에서 다른 2개 VM보다 효율성이 낮은 것으로 보이므로 TraceMonkey의 성능 향상은 작거나 전혀 없다. 재귀를 광범위하게 사용하는 벤치마크는 3d-cube, 3draytrace, access-binary-trees, string-tagcloud이며, 이어서 상세 성능 지표를 살펴본다. 그림 11에는 fraccontrolflow-recursive를 제시한다.
해석된 명령어의 비율과 네이티브 코드로 실행된 명령어의 비율이다.
• 현재 구현은 eval와 일부 항목을 트레이스하지 않는다. 이 그림은 많은 프로그램에서 거의 모든 코드를 네이티브 방식으로 실행할 수 있음을 보여주며, 다른 함수들은 C로 구현된다. tofte와 date-format-xparb는 메인 루프에서 그러한 함수를 사용하므로 트레이스하지 않는다. 그림 12는 전체 실행 시간을 4가지 활동으로 구분한다. 활동은 기록하지 않는 동안 바이트코드를 해석하는 것, 트레이스를 기록하는 것 •, 그리고 컴파일하는 것이다. 현재 구현은 정규식 replace 연산을 통과하여 트레이스하지 않는다(기록된 트레이스를 해석하는 데 걸린 시간 포함).
replace 함수에는 네이티브 코드로 컴파일된 트레이스와 네이티브 코드 트레이스를 실행하는 함수를 전달할 수 있다. 이 함수 객체는 대체 텍스트를 계산하는 데 사용된다.
이러한 상세 지표를 통해 트레이스 성능의 단순한 모델에 필요한 매개변수를 추정할 수 있다. 현재 구현은 호출된 함수를 트레이스하지 않는다.
이 추정치는 replace 함수와 마찬가지로 매우 대략적인 것으로 간주해야 한다.
string-unpack-code의 실행 시간은 개별 항목에서 관찰된 값과 마찬가지로 매우 대략적인 것으로 간주되며, 이러한 replace 호출이 실행 시간의 대부분을 차지한다.

- Two programs trace well, but have a long compilation time. `access-nbody` forms a large number of traces (81). `crypto-md5` forms one very long trace. We expect to improve performance on this programs by improving the compilation speed of nanojit.
- Some programs trace very well, and speed up compared to the interpreter, but are not as fast as SFX and/or V8, namely `bitops-bits-in-byte`, `bitops-nsieve-bits`, `access-fannkuch`, `access-nsieve`, and `crypto-aes`. The reason is not clear, but all of these programs have nested loops with small bodies, so we suspect that the implementation has a relatively high cost for calling nested traces. `string-fasta` traces well, but its run time is dominated by string processing builtins, which are unaffected by tracing and seem to be less efficient in SpiderMonkey than in the two other VMs.

Detailed performance metrics. In Figure 11 we show the fraction of instructions interpreted and the fraction of instructions executed as native code. This figure shows that for many programs, we are able to execute almost all the code natively.

Figure 12 breaks down the total execution time into four activities: interpreting bytecodes while not recording, recording traces (including time taken to interpret the recorded trace), compiling traces to native code, and executing native code traces.

These detailed metrics allow us to estimate parameters for a simple model of tracing performance. These estimates should be considered very rough, as the values observed on the individual benchmarks have large standard deviations (on the order of the

	Loops	Trees	Traces	Aborts	Flushes	Trees/Loop	Traces/Tree	Traces/Loop	Speedup
3d-cube	25	27	29	3	0	1.1	1.1	1.2	2.20x
3d-morph	5	8	8	2	0	1.6	1.0	1.6	2.86x
3d-raytrace	10	25	100	10	1	2.5	4.0	10.0	1.18x
access-binary-trees	0	0	0	5	0	-	-	-	0.93x
access-fannkuch	10	34	57	24	0	3.4	1.7	5.7	2.20x
access-nbody	8	16	18	5	0	2.0	1.1	2.3	4.19x
access-nsieve	3	6	8	3	0	2.0	1.3	2.7	3.05x
bitops-3bit-bits-in-byte	2	2	2	0	0	1.0	1.0	1.0	25.47x
bitops-bits-in-byte	3	3	4	1	0	1.0	1.3	1.3	8.67x
bitops-bitwise-and	1	1	1	0	0	1.0	1.0	1.0	25.20x
bitops-nsieve-bits	3	3	5	0	0	1.0	1.7	1.7	2.75x
controlflow-recursive	0	0	0	1	0	-	-	-	0.98x
crypto-aes	50	72	78	19	0	1.4	1.1	1.6	1.64x
crypto-md5	4	4	5	0	0	1.0	1.3	1.3	2.30x
crypto-sha1	5	5	10	0	0	1.0	2.0	2.0	5.95x
date-format-tofte	3	3	4	7	0	1.0	1.3	1.3	1.07x
date-format-xparb	3	3	11	3	0	1.0	3.7	3.7	0.98x
math-cordic	2	4	5	1	0	2.0	1.3	2.5	4.92x
math-partial-sums	2	4	4	1	0	2.0	1.0	2.0	5.90x
math-spectral-norm	15	20	20	0	0	1.3	1.0	1.3	7.12x
regexp-dna	2	2	2	0	0	1.0	1.0	1.0	4.21x
string-base64	3	5	7	0	0	1.7	1.4	2.3	2.53x
string-fasta	5	11	15	6	0	2.2	1.4	3.0	1.49x
string-tagcloud	3	6	6	5	0	2.0	1.0	2.0	1.09x
string-unpack-code	4	4	37	0	0	1.0	9.3	9.3	1.20x
string-validate-input	6	10	13	1	0	1.7	1.3	2.2	1.86x

벤치마크는 큰 표준편차를 보인다(대략 루프 트리 트레이스 중단 플러시 트리/루프 트레이스/트리 트레이스/루프 성능 향상 3d-cube 25 27 29 3 0 1.1 1.1 1.2 2.20배 3d-morph 5 8 8 2 0 1.6 1.0 1.6 2.86배 3d-raytrace 10 25 100 10 1 2.5 4.0 10.0 1.18배 access-binary-trees 0 0 0 5 0 - - - 0.93배 access-fannkuch 10 34 57 24 0 3.4 1.7 5.7 2.20배 access-nbody 8 16 18 5 0 2.0 1.1 2.3 4.19배 access-nsieve 3 6 8 3 0 2.0 1.3 2.7 3.05배 bitops-3bit-bits-in-byte 2 2 2 0 0 1.0 1.0 1.0 25.47배 bitops-bits-in-byte 3 3 4 1 0 1.0 1.3 1.3 8.67배 bitops-bitwise-and 1 1 1 0 0 1.0 1.0 1.0 25.20배 bitops-nsieve-bits 3 3 5 0 0 1.0 1.7 1.7 2.75배 controlflow-recursive 0 0 0 1 0 - - - 0.98배 crypto-aes 50 72 78 19 0 1.4 1.1 1.6 1.64배 crypto-md5 4 4 5 0 0 1.0 1.3 1.3 2.30배 crypto-sha1 5 5 10 0 0 1.0 2.0 2.0 5.95배 date-format-tofte 3 3 4 7 0 1.0 1.3 1.3 1.07배 date-format-xparb 3 3 11 3 0 1.0 3.7 3.7 0.98배 math-cordic 2 4 5 1 0 2.0 1.3 2.5 4.92배 math-partial-sums 2 4 4 1 0 2.0 1.0 2.0 5.90배 math-spectral-norm 15 20 20 0 0 1.3 1.0 1.3 7.12배 regexp-dna 2 2 2 0 0 1.0 1.0 1.0 4.21배 string-base64 3 5 7 0 0 1.7 1.4 2.3 2.53배 string-fasta 5 11 15 6 0 2.2 1.4 3.0 1.49배 string-tagcloud 3 6 6 5 0 2.0 1.0 2.0 1.09배 string-unpack-code 4 4 37 0 0 1.0 9.3 9.3 1.20배 string-validate-input 6 10 13 1 0 1.7 1.3 2.2 1.86배

Figure 13. Detailed trace recording statistics for the SunSpider benchmark set.

그림 13. SunSpider 벤치마크 세트의 상세한 트레이스 기록 통계.

mean). We exclude `regexp-dna` from the following calculations, because most of its time is spent in the regular expression matcher, which has much different performance characteristics from the other programs. (Note that this only makes a difference of about 10% in the results.) Dividing the total execution time in processor clock cycles by the number of bytecodes executed in the base interpreter shows that on average, a bytecode executes in about 35 cycles. Native traces take about 9 cycles per bytecode, a 3.9x speedup over the interpreter.

Using similar computations, we find that trace recording takes about 3800 cycles per bytecode, and compilation 3150 cycles per bytecode. Hence, during recording and compiling the VM runs at 1/200 the speed of the interpreter. Because it costs 6950 cycles to compile a bytecode, and we save 26 cycles each time that code is run natively, we break even after running a trace 270 times.

The other VMs we compared with achieve an overall speedup of 3.0x relative to our baseline interpreter. Our estimated native code speedup of 3.9x is significantly better. This suggests that our compilation techniques can generate more efficient native code than any other current JavaScript VM.

These estimates also indicate that our startup performance could be substantially better if we improved the speed of trace recording and compilation. The estimated 200x slowdown for recording and compilation is very rough, and may be influenced by startup factors in the interpreter (e.g., caches that have not warmed up yet during recording). One observation supporting this conjecture is that in the tracer, interpreted bytecodes take about 180 cycles to run. Still, recording and compilation are clearly both expensive, and a better implementation, possibly including redesign of the LIR abstract syntax or encoding, would improve startup performance.

Our performance results confirm that type specialization using trace trees substantially improves performance. We are able to outperform the fastest available JavaScript compiler (V8) and the fastest available JavaScript inline threaded interpreter (SFX) on 9 of 26 benchmarks.

다음 계산에서는 regexp-dna을 제외한다. 실행 시간 대부분을 정규 표현식 매처에서 보내기 때문이다. 가장 빠른 JavaScript 인라인 스레드 인터프리터(SFX)는 26개 벤치마크 중 9개에서 다른 프로그램과는 성능 특성이 크게 다르다. (이로 인한 결과의 차이는 약 10%에 불과하다는 점에 유의한다.) 총 실행 시간을 프로세서 8. 관련 연구 클록 사이클 수로 나눈 뒤 기본 인터프리터에서 실행된 바이트코드 수로 다시 나누면 바이트코드 하나를 실행하는 데 평균 약 35사이클이 걸림을 알 수 있다. 동적 언어를 위한 트레이스 최적화. 가장 근접한 영역은 35사이클이다. 네이티브 트레이스는 바이트코드당 약 9사이클이 걸려 인터프리터보다 3.9배 빠르며, 관련 연구에서는 트레이스 최적화를 적용하여 타입 특수화를 수행한다. 동적 언어이다. 기존 연구도 인터프리터를 따라 가드를 배치하여 타입 특수화된 코드를 추측적으로 생성한다는 아이디어를 공유한다. 유사한 방식으로 계산하면 트레이스 기록에는 바이트코드당 약 3800사이클이, 컴파일에는 트레이스당 3150사이클이 걸린다.

8. Related Work

Trace optimization for dynamic languages. The closest area of related work is on applying trace optimization to type-specialize dynamic languages. Existing work shares the idea of generating type-specialized code speculatively with guards along interpreter traces.

To our knowledge, Rigo's Psyco (16) is the only published type-specializing trace compiler for a dynamic language (Python). Psyco does not attempt to identify hot loops or inline function calls. Instead, Psyco transforms loops to mutual recursion before running and traces all operations.

Pall's LuaJIT is a Lua VM in development that uses trace compilation ideas. (1). There are no publications on LuaJIT but the creator has told us that LuaJIT has a similar design to our system, but will use a less aggressive type speculation (e.g., using a floating-point representation for all number values) and does not generate nested traces for nested loops.

General trace optimization. General trace optimization has a longer history that has treated mostly native code and typed languages like Java. Thus, these systems have focused less on type specialization and more on other optimizations.

Dynamo (7) by Bala et al, introduced native code tracing as a replacement for profile-guided optimization (PGO). A major goal was to perform PGO online so that the profile was specific to the current execution. Dynamo used loop headers as candidate hot traces, but did not try to create loop traces specifically.

Trace trees were originally proposed by Gal et al. (11) in the context of Java, a statically typed language. Their trace trees actually inlined parts of outer loops within the inner loops (because

바이트코드. 따라서 기록 및 컴파일 중 VM은 인터프리터 속도의 1/200로 실행된다. 우리가 아는 한 Rigo의 Psyco (16)는 출판된 유일한 동적 언어(Python)를 위한 타입 특수화 트레이스 컴파일러이다. 바이트코드 컴파일에는 6950사이클이 들기 때문이다. 바이트코드를 컴파일하고 해당 코드가 네이티브로 실행될 때마다 26사이클을 절약한다. Psyco는 핫 루프를 식별하거나 함수 호출을 인라인하려 하지 않는다. 트레이스를 네이티브로 270회 실행하면 손익분기점에 도달한다. 대신 Psyco는 실행 전에 루프를 상호 재귀로 변환하고 모든 연산을 트레이스한다. 비교 대상인 다른 VM들은 전반적인 성능 향상을 달성한다. 기준 인터프리터 대비 3.0배이다. 네이티브 코드의 추정 성능 향상인 3.9배는 훨씬 우수하다. Pall의 LuaJIT은 트레이스 컴파일 아이디어를 사용하는 개발 중인 Lua VM이다. 이는 컴파일 아이디어를 시사한다. (1). LuaJIT에 관한 출판물은 없지만, 개발자는 LuaJIT의 설계가 우리 시스템과 유사하다고 알려 왔다. 우리의 컴파일 기법은 현재의 다른 어떤 JavaScript VM보다도 효율적인 네이티브 코드를 생성할 수 있다. 다만 LuaJIT은 덜 공격적인 타입 추측을 사용하고(예를 들어 모든 숫자 값에 부동 소수점 표현을 사용함) 중첩 루프에 대한 중첩 트레이스를 생성하지 않는다. 또한 이 추정치는 트레이스 기록 속도를 개선하면 시작 성능을 상당히 높일 수 있음을 보여 준다. 그리고 컴파일. 기록 시 추정되는 200배의 성능 저하. 일반적인 트레이스 최적화. 일반적인 트레이스 최적화와 컴파일에 대한 추정치는 매우 대략적이며, 인터프리터의 시작 요인(예를 들어 기록 중 아직 워밍업되지 않은 캐시)의 영향을 받을 수 있다. 일반적인 트레이스 최적화는 주로 네이티브 코드와 Java 같은 정적 타입 언어를 다루어 온 오랜 역사가 있다. 따라서 이러한 시스템은 타입에 덜 초점을 맞춘다. 이 추측을 뒷받침하는 한 가지 관찰은 특수화보다 다른 최적화에 더 초점을 맞춘다는 것이다. 트레이스 기록기에서 해석된 바이트코드를 실행하는 데 약 180사이클이 걸린다. 그럼에도 기록과 컴파일은 둘 다 분명히 비용이 많이 든다. Bala 등이 개발한 Dynamo (7)는 프로파일 유도 최적화(PGO)를 대체하는 더 나은 방법으로 네이티브 코드 추적을 도입했다. 주요 구현 목표는 프로파일이 현재 실행에 특화되도록 PGO를 온라인으로 수행하는 것이었다. LIR 추상 구문이나 인코딩의 재설계를 포함할 수 있는 더 나은 구현은 시작 성능을 개선할 것이다. 현재 실행. Dynamo는 루프 헤더를 후보 핫 트레이스로 사용했지만, 루프 트레이스를 특별히 생성하려 하지는 않았다. 우리의 성능 결과는 트레이스를 사용한 타입 특수화가 효과적임을 확인한다. 트레이스 트리는 성능을 크게 향상시킨다. 우리는 현재 사용 가능한 가장 빠른 JavaScript 컴파일러(V8)보다 뛰어난 성능을 낼 수 있다. 트레이스 트리는 원래 Gal 등이 (11) 정적 타입 언어인 Java의 맥락에서 제안했다.

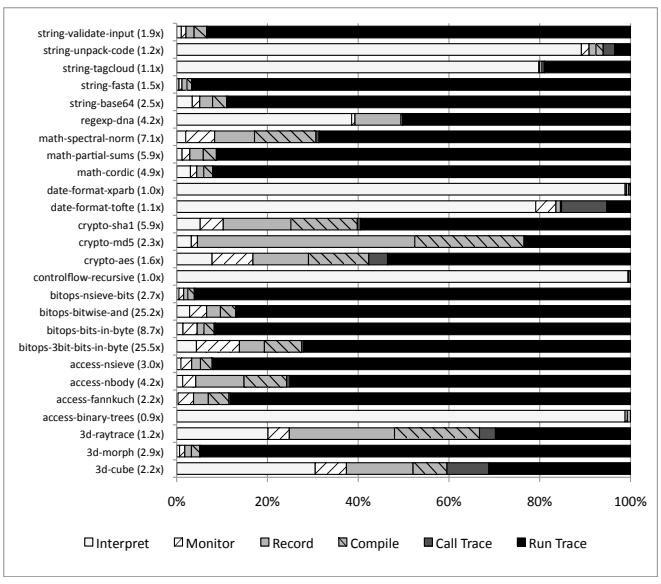


Figure 12. Fraction of time spent on major VM activities. The speedup vs. interpreter is shown in parentheses next to each test. Most programs where the VM spends the majority of its time running native code have a good speedup. Recording and compilation costs can be substantial; speeding up those parts of the implementation would improve SunSpider performance.

9. 결론 그림 12. 주요 VM 활동에 소요된 시간의 비율. 본 논문은 동적 언어를 효율적으로 실행하는 방법을 설명했으며, 인터프리터 대비 성능 향상은 각 테스트 옆의 괄호 안에 표시한다. 핫 트레이스를 기록하고 타입 특수화된 네이티브 코드를 생성한다. VM이 대부분의 시간을 네이티브 코드 실행에 쓰는 프로그램은 대체로 성능 향상이 크다. 우리의 기법은 적극적으로 인라인화된 루프에 초점을 맞추며, 각 기록 및 컴파일 비용은 상당할 수 있다. 각 루프에 대해 런타임에 관찰된 루프 내 경로와 값 타입을 나타내는 네이티브 코드 트레이스 트리를 생성한다. 구현의 해당 부분을 고속화하면 SunSpider 성능이 개선될 것이다. 우리는 루프 중첩 관계를 식별하고 중첩된 트레이스를 생성하여 루프 중첩을 통과하는 다수의 경로로 인한 과도한 코드 중복을 방지하는 방법을 설명했다.

erate native code with nearly the same structure but better performance.

Call threading, also known as context threading (8), compiles methods by generating a native call instruction to an interpreter method for each interpreter bytecode. A call-return pair has been shown to be a potentially much more efficient dispatch mechanism than the indirect jumps used in standard bytecode interpreters.

Inline threading (15) copies chunks of interpreter native code which implement the required bytecodes into a native code cache, thus acting as a simple per-method JIT compiler that eliminates the dispatch overhead.

Neither call threading nor inline threading perform type specialization.

Apple's SquirrelFish Extreme (5) is a JavaScript implementation based on call threading with selective inline threading. Combined with efficient interpreter engineering, these threading techniques have given SFX excellent performance on the standard SunSpider benchmarks.

Google's V8 is a JavaScript implementation primarily based on inline threading, with call threading only for very complex operations.

이들의 트레이스 트리는 실제로 내부 루프 안에 외부 루프의 일부를 인라인했다(거의 동일한 구조이면서도 더 나은 성능을 내는 네이티브 코드를 생성하기 때문이다). 성능. 컨텍스트 스레딩이라고도 하는 호출 스레딩 (8)은 인터프리터로의 네이티브 호출 명령어를 생성하여 메서드를 컴파일하며, 각 인터프리터 바이트코드마다 하나의 인터프리터 메서드를 사용한다. 호출-반환 쌍은 잠재적으로 훨씬 더 효율적인 디스패치 메커니즘인 것으로 밝혀졌다. 이는 표준 바이트코드 인터프리터에서 사용하는 간접 점프보다 효율적이다. 인라인 스레딩 (15)은 인터프리터 네이티브 코드의 일부를 복사한다. 이 코드는 필요한 바이트코드를 구현하며 네이티브 코드 캐시에 복사된다. 따라서 디스패치 오버헤드를 제거하는 단순한 메서드별 JIT 컴파일러로 작동한다. 디스패치 오버헤드. 호출 스레딩과 인라인 스레딩은 모두 타입 특수화를 수행하지 않는다. Apple의 SquirrelFish Extreme (5)은 JavaScript 구현체이며, 선택적 인라인 스레딩과 함께 호출 스레딩을 기반으로 한다. 결합하면, 효율적인 인터프리터 엔지니어링과 결합된 이러한 스레딩 기법은 SFX가 표준 Sun- Spider 벤치마크에서 뛰어난 성능을 발휘하게 했다. Google의 V8은 주로 인라인 스레딩을 기반으로 하는 JavaScript 구현체이며, 호출 스레딩은 매우 복잡한 경우에만 사용한다. 연산.

9. Conclusions

This paper described how to run dynamic languages efficiently by recording hot traces and generating type-specialized native code. Our technique focuses on aggressively inlined loops, and for each loop, it generates a tree of native code traces representing the paths and value types through the loop observed at run time. We explained how to identify loop nesting relationships and generate nested traces in order to avoid excessive code duplication due to the many paths through a loop nest. We described our type specialization algorithm. We also described our trace compiler, which translates a trace from an intermediate representation to optimized native code in two linear passes.

타입 특수화 알고리즘을 설명했다.

또한 중간 표현의 트레이스를 최적화된 네이티브 코드로 2번의 선형 패스를 통해 변환하는 트레이스 컴파일러를 설명했다. 내부 루프가 먼저 핫해짐에 따라 훨씬 더 심한 꼬리 복제

Our experimental results show that in practice loops typically are entered with only a few different combinations of value types of variables. Thus, a small number of traces per loop is sufficient to run a program efficiently. Our experiments also show that on programs amenable to tracing, we achieve speedups of 2x to 20x.

가 발생한다. 실험 결과, 실제로 루프에 진입할 때 변수 값의 타입 조합은 대개 몇 가지뿐인 것으로 나타났다. Zaleski et al.의 YETI (19)는 인라이닝과 간접 점프 제거를 달성하기 위해 Dynamo 방식의 트레이싱을 Java에 적용했다.

따라서 루프당 소수의 트레이스만으로도 충분하며, 기타 최적화도 적용할 수 있다. 그들의 주된 초점은 프로그램을 효율적으로 실행할 수 있는 인터프리터를 설계하는 것이었다.

또한 실험 결과, 트레이싱에 적합한 프로그램에서는 2배에서 20배의 성능 향상을 달성했다. 이 인터프리터는 쉽게 점진적으로 재설계하여 트레이싱 방식으로 전환할 수 있었다.

10. Future Work

Work is underway in a number of areas to further improve the performance of our trace-based JavaScript compiler. We currently do not trace across recursive function calls, but plan to add the support for this capability in the near term. We are also exploring adoption of the existing work on tree recompilation in the context of the presented dynamic compiler in order to minimize JIT pause times and obtain the best of both worlds, fast tree stitching as well as the improved code quality due to tree recompilation.

We also plan on adding support for tracing across regular expression substitutions using lambda functions, function applications and expression evaluation using `eval`. All these language constructs are currently executed via interpretation, which limits our performance for applications that use those features.

10. 향후 연구 여러 메서드 중 일부를 얼마든지 포함할 수 있다. 따라서 컴파일러는 더 큰 유연성을 가지며 잠재적으로 더 나은 코드를 생성할 수 있다. 트레이스 기반 JavaScript 컴파일러의 성능을 더욱 개선하기 위한 연구가 여러 영역에서 진행 중이다. 현재 프로파일링 및 컴파일 시스템은 그에 상응하여 더 복잡하다. 현재 재귀 함수 호출을 가로질러 트레이스하지 않지만, 이 기능을 지원할 계획이다. 가까운 시일 내에 이 기능을 지원할 예정이다. 또한 동적 언어를 위한 타입 특수화를 살펴보고 있다. 동적 언어 구현자들은 성능을 위한 타입 특수화의 중요성을 오래전부터 인식해 왔다. 또한 JIT 일시 중지 시간을 최소화하기 위해 제시한 동적 컴파일러의 맥락에서 기존 트리 재컴파일 연구를 도입하는 방안을 살펴보고 있다. 이전 연구는 대부분 트레이스가 아니라 메서드에 초점을 맞추었다. 이를 통해 빠른 트리 연결과 짧은 처리 시간을 모두 얻을 수 있다. 트리 재컴파일로 인한 코드 품질 향상도 얻을 수 있다. Chambers et. al (9)은 Self에서 입력 타입에 맞게 특수화된 프로시저의 여러 버전을 컴파일한다는 아이디어를 개척했다. 또한 람다 함수를 사용하는 정규식 치환을 가로질러 트레이스하는 기능을 지원할 계획이다. 한 구현에서는 특수화된 메서드를 생성했다. 또한 함수 적용과 eval을 이용한 표현식 평가도 지원할 계획이다. 새로운 입력 타입으로 메서드가 호출될 때마다 온라인으로 특수화된 메서드를 생성했다. 이러한 언어 구성 요소는 모두 현재 인터프리터를 통해 실행되므로 해당 기능을 사용하는 애플리케이션의 성능이 제한된다. 다른 구현에서는 오프라인 전역 프로그램 정적 분석을 사용하여 호출 지점의 입력 타입과 상수 수신자 타입을 추론했다.

YETI, from Zaleski et al. (19) applied Dynamo-style tracing to Java in order to achieve inlining, indirect jump elimination, and other optimizations. Their primary focus was on designing an interpreter that could easily be gradually re-engineered as a tracing VM.

Suganuma et al. (18) described region-based compilation (RBC), a relative of tracing. A region is an subprogram worth optimizing that can include subsets of any number of methods. Thus, the compiler has more flexibility and can potentially generate better code, but the profiling and compilation systems are correspondingly more complex.

Suganuma et al. (18)은 트레이싱과 유사한 영역 기반 컴파일(RBC)을 설명했다. 영역은 최적화할 가치가 있는 하위 프로그램이며

Type specialization for dynamic languages. Dynamic language implementors have long recognized the importance of type specialization for performance. Most previous work has focused on methods instead of traces.

Chambers et. al (9) pioneered the idea of compiling multiple versions of a procedure specialized for the input types in the language Self. In one implementation, they generated a specialized method online each time a method was called with new input types. In another, they used an offline whole-program static analysis to infer input types and constant receiver types at call sites. Interestingly, the two techniques produced nearly the same performance.

Salib (17) designed a type inference algorithm for Python based on the Cartesian Product Algorithm and used the results to specialize on types and translate the program to C++.

흥미롭게도 두 기법의 성능은 거의 동일했다. 감사의 글 Salib (17)은 Cartesian Product Algorithm에 기반한 Python용 타입 추론 알고리즘을 설계하고, 그 결과를 이용해 타입을 특수화하고 프로그램을 C++로 변환했다. 이 연구의 일부는 National Science Foundation의 지원을 받았다.

McCloskey (14) has work in progress based on a language-independent type inference that is used to generate efficient C implementations of JavaScript and Python programs.

McCloskey (14)는 JavaScript 및 Python 프로그램의 효율적인 C 구현을 생성하는 데 사용되는 언어 독립적 타입 추론을 기반으로 한 연구를 진행 중이다.

JavaScript 및 Python 프로그램의 구현.

Native code generation by interpreters. The traditional interpreter design is a virtual machine that directly executes ASTs or machine-code-like bytecodes. Researchers have shown how to gen-

미국 정부는 인터프리터에 의한 네이티브 코드 생성을 복제하고 배포할 권한이 있다.

저작권 표시와 관계없이 정부 목적을 위한 전통적인 재인쇄 인터프리터 설계는 AST 또는 그 위의 주석을 직접 실행하는 가상 머신이다.

모든 의견, 조사 결과 및 결론이나 권고 사항, 그리고 기계어와 유사한 바이트코드.

Acknowledgments

Parts of this effort have been sponsored by the National Science Foundation under grants CNS-0615443 and CNS-0627747, as well as by the California MICRO Program and industrial sponsor Sun Microsystems under Project No. 07-127.

CNS-0615443 및 CNS-0627747 연구비와 California MICRO Program, 그리고 산업 후원사인 Sun Microsystems의 Project No. 07-127 지원을 받았다.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Any opinions, findings, and conclusions or recommendations expressed here are those of the author and should

not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of the National Science foundation (NSF), any other agency of the U.S. Government, or any of the companies mentioned above.

연구자들은 방법을 보였다. 여기에 표현된 견해는 저자의 것이며 공식적인 견해를 반드시 대변하는 것으로 해석해서는 안 된다,
[10] A. Gal. 가상 머신에서의 효율적인 바이트코드 검증 및 컴파일에 관한 박사학위논문. University Of California, Irvine 박사학위논문,
명시적이든 묵시적이든 Na-의 정책이나 보증, 2006. tional Science foundation (NSF), 그 밖의 모든 미국 정부 기관 또는 위에서 언급한 모든 기업.
[11] A. Gal, C. W. Probst, M. Franz 공저. HotpathVM: 자원이 제한된 장치를 위한 효과적인 JIT 컴파일러.

References

[1] LuaJIT roadmap 2008 - http://lua-users.org/lists/lua-l/2008-02/msg00051.html.

[2] Mozilla — Firefox web browser and Thunderbird email client - http://www.mozilla.com.

[3] SPECJVM98 - http://www.spec.org/jvm98/.

International Conference on Virtual Execution Environments 논문집, 페이지 참고문헌 144-153. ACM Press, 2006. [1] LuaJIT 로드맵 2008 -
http://lua-users.org/lists/lua-l/2008- [12] C. Garrett, J. Dean, D. Grove, and C. Chambers. 측정 및 02/msg00051.html. 동적 수신자 클래스 분포의 적용. 1994. [2]
Mozilla — Firefox 웹 브라우저 및 Thunderbird 이메일 클라이언트 - [13] J. Ha, M. R. Haghighat, S. Cong, and K. S. McKinley. 동시 실행형 http://www.mozilla.com.
JavaScript를 위한 트레이스 기반 적시 컴파일러. 컴퓨터과학부 [3] SPECJVM98 - http://www.spec.org/jvm98/. 과학부, The University of Texas at Austin, 기술 보고서
TR-09-06, 2009.

[4] SpiderMonkey (JavaScript-C) Engine - http://www.mozilla.org/js/spidermonkey/.

[5] Surfin' Safari - Blog Archive - Announcing SquirrelFish Extreme - http://webkit.org/blog/214/introducing-squirrelfish-extreme/.

[6] A. Aho, R. Sethi, J. Ullman, and M. Lam. Compilers: Principles, techniques, and tools, 2006.

[4] SpiderMonkey (JavaScript-C) 엔진 - [14] B. McCloskey. 개인 서신. http://www.mozilla.org/js/spidermonkey/. [15] I. Piumarta와 F. Riccardi 공저. 선택적
인라이닝을 통한 직접 스레드 코드 최적화- [5] Surfin' Safari - 블로그 아카이브 - SquirrelFish Extreme 발표 - 선택적 인라이닝. ACM SIGPLAN 1998 학술대회 논문집
http://webkit.org/blog/214/introducing-squirrelfish-extreme/. 프로그래밍 언어 설계 및 구현, 페이지 291- [6] A. Aho, R. Sethi, J. Ullman, and M. Lam.
컴파일러: 원리, 300. ACM, New York, NY, USA에서 1998년에 발행. 기법 및 도구, 2006. [16] A. Rigo.

[7] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2000.

표현 기반 적시 특수화 및 [7] V. Bala, E. Duesterwald, and S. Banerjia.

[8] M. Berndl, B. Vitale, M. Zaleski, and A. Brown. Context Threading: a Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 15–26, 2005.

[9] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed O bject-Oriented Programming Language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 146–160. ACM New York, NY, USA, 1989.

[10] A. Gal. *Efficient Bytecode Verification and Compilation in a Virtual Machine Dissertation*. PhD thesis, University Of California, Irvine, 2006.

[11] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 144–153. ACM Press, 2006.

[12] C. Garrett, J. Dean, D. Grove, and C. Chambers. Measurement and Application of Dynamic Receiver Class Distributions. 1994.

[13] J. Ha, M. R. Haghighat, S. Cong, and K. S. McKinley. A concurrent trace-based just-in-time compiler for javascript. Dept.of Computer Sciences, The University of Texas at Austin, TR-09-06, 2009.

[14] B. McCloskey. Personal communication.

[15] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 291–300. ACM New York, NY, USA, 1998.

[16] A. Rigo. Representation-Based Just-In-time Specialization and the Psyco Prototype for Python. In *PEPM*, 2004.

[17] M. Salib. Starkiller: A Static Type Inferencer and Compiler for Python. In *Master's Thesis*, 2004.

[18] T. Suganuma, T. Yasue, and T. Nakatani. A Region-Based Compilation Technique for Dynamic Compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(1):134–174, 2006.

[19] M. Zaleski, A. D. Brown, and K. Stoodley. YETI: A graduallY Extensible Trace Interpreter. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 83–93. ACM Press, 2007.

Dynamo: Python을 위한 투명한 Psyco 프로토타입. PEPM, 2004. 동적 최적화 시스템. ACM SIGPLAN 논문집 [17] M. Salib. Starkiller: 프로그래밍 언어 설계 및 구현
학술대회를 위한 정적 타입 추론기 및 컴파일러, Python. 석사학위논문, 2004. 페이지 1-12. ACM Press, 2000. 영역 기반 컴파일- [8] M. Berndl, B. Vitale, M. Zaleski, A.
Brown. 컨텍스트 스레딩: 동적 컴파일러를 위한 tion 기법. ACM Transactions on Proa 가상 머신 Ingramming Languages and Systems (TOPLAS)를 위한 유연하고
효율적인 디스패치 기법, 28(1):134-174, 2006. 인터프리터. Code Generation and Optimization, 2005에 수록. YETI: 점진적Y 국제 심포지엄, 15-26쪽, 2005. 확장
가능한 트레이스 인터프리터. 국제 학술대회 논문집에 수록 [9] C. Chambers, D. Ungar. 맞춤화: 가상 실행 환경에 관한 최적화 컴파일러 학회, 83-93쪽. ACM 동적 타입 객체
지향 Pro-인 SELF를 위한 기술 Press, 2007. 프로그래밍 언어.

ACM SIGPLAN 1989 프로그래밍 언어 설계 및 구현 학술대회 논문집, 146-160쪽. ACM, New York, NY, USA, 1989년.